促进软件开发领域知识与创新的传播

架构师

ARCHITECT

热点 | Hot

Rust 1.0发布一周年，发展回顾与总结

推荐文章 | Article

跨过技术创业的那些坎儿

观点 | Opinion

开源是物联网开发消除厂商绑定的关键

现在Google制造自己的芯片，Intel要发疯

特别专栏 | Column

Kafka的崛起微服务架构终极探讨

Geekbang 极客邦科技 | InfoQ

2016年7月

卷首语：架构师的能力与责任

我大学毕业当了 IT 民工，一年之后，拿到了助理工程师职称证书，顿觉自己成了有用之身，大有用武之地，传说中的高工就在不远处等着我，那时候还不知道什么是架构师。而今的技术圈里，架构师似乎才是高大上的代名词，毕竟不是每个公司都有研究员、领域专家、科学家的。

架构师与工程师相比，有多大不同呢？在我看来，架构师没有什么特殊的，只是工程师在技术路径的进一步延伸。架构师，是随着系统规模越来越大，越来越复杂而衍生出的角色。许多公司里并没有固定的架构师职位，但具备了相应能力，获得了团队认可，在项目中承担了架构职责，你就是架构师。

架构师的能力大，责任更大，我对架构师的职责定义如下：

- 以工程思维全面理解业务需求；
- 基于模型和基础模式抽象简化；
- 提出恰当可行的整体解决方案；
- 在限定资源范围完成明确目标；
- 满足业务需求且保证系统质量；
- 在可预见的周期内具备扩展性；
- 并在系统生命周期内持续演进。

所以架构师要靠项目实践积累经验，并结合系统化的学习，提升自身能力。知易行难，架构师是很难培训出来的，多数都是身经百战，方百炼成钢。即便如此，也很难在具体项目中知行合一。工作中架构师是技术方面负责人，遇到问题多数靠自己解决，没人能传帮带，所以架构师必须具备强悍的自学能力和毫不松懈的自我驱动力，很多时候，凭的就是心中那一口气。

架构师的责任心也很重要，因为架构方面的工作往往处于重要但不紧急的尴尬境地，如果架构师在这方面自己不重视，那还怎么能做好呢？当然，要是只关注技术架构，不关注业务目标，就更不合格了，项目组的每一个成员都需要理解业务目标，并为之努力。

温伯格曾说：“一个系统，就是对世界的一种看法。”世界是什么样的？多元、有机、不断变化、并不完美。架构师设计、搭建、维护系统，这个过程，创造了一个小世界。而这个小小的世界，融合了架构师对大千世界的体悟和取舍，是每个架构师的智慧与汗水的结晶。

架构师，是一种修炼，更是一种修行，是一种别样的人生。

当当架构部总监 **史海峰**

CONTENTS / 目录

架构师

2016 年 7 月刊

本期主编 郭 蕾

流程编辑 丁晓昀

发行人 霍泰稳

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

CNUTCon 2016
全球容器技术大会

2016.9.9－10　北京 喜来登长城饭店

案 / 例 / 剖 / 析 · 实 / 践 / 驱 / 动

8折优惠（截止8月7日）团购报名更多优惠

主办方 Geekbang. 极客邦科技　InfoQ

最佳的容器实践案例都在这里

10大精彩议题

- 容器云专场
- 持续集成/持续交付
- 微服务
- Kubernetes实践
- Mesos实践
- Docker实践
- 金融场
- 创业场
- 电商场
- 综合场

超40位大牛讲师
解读最新、最前沿的容器技术

徐振中
Netflix
资深构架师

李 响
CoreOS
分布式项目主管

陈东洛
Docker
Swarm负责人

Timothy Chen
Mesosphere
分布式系统专家

联系我们 / CONTACT US

商务咨询：yolanda@infoq.com　13426412029

会务咨询：cici@infoQ.com　15810393465

票务咨询：abby@geekbang.org　15801572605

扫码进入聊聊架构社群

扫码进入容器大会官网

为什么说开源是物联网开发消除厂商绑定的关键

作者 Olivier Pauzet 译者 大愚若智

物联网（IoT）项目的开发可能是一个艰巨的任务。从原型的设计和代码的构建，到产品的最终发布和全球部署，如何确保您的 IoT 项目顺利实现和上市？

鉴于未来数年将有数十亿设备投放市场，封闭的专属系统会使不同设备间的互操作变得更困难。ProgrammableWeb 主编 David Berlind 将持续演化的封闭 IoT 生态系统看作一种“连环事故”。让按照设计本无法配合使用的不同系统组件和元素协同工作，可能需要花费大量的时间和精力，延长部署所需时间并导致总成本飙升。开源技术是解决这种问题让不同设备相互通信的方法之一。

不同应用领域的 IoT 解决方案有很多共通之处：需要无线连接的能力，需要在设备和后端系统之间具备通信能力，需要收集和解析获得的数据，大量共通之处不一而足。但是随之而来的大量专有系统通常是“各自为政”的，这使得解决方案的开发和构建工作变得更为复杂，需要额外投入更多

时间。专有系统还会让不同系统之间的开放式通信变得更复杂，并有可能妨碍到未来的创新和更广泛的运用。

面对这个快速演进，碎片化程度日趋严重的行业，这些挑战也变得越来越棘手，不过只要愿意配合使用，市面上依然有几个可用的解决方案。之所以说开源技术是 IoT 开发的关键，主要有下列几个原因。

用开放式协作和标准铺平道路

行业标准的建立和实施可以帮助我们实现更强的互操作性。通过合作制定的完善标准可提供更丰富的选择和更大灵活性：开发者可以针对具体需求，使用不同供应商的设备构建解决方案，进而在构建解决方案的过程中实现更大的创新，获得更大成本效益。

除了标准化开发，另一种互补的方法是将行业生态系统制定的设计和规范开放给开源社区，借此形成由所有人遵守的开源硬件和接口标准。这种方法日益受到欢迎，随着主要业内人士通过合作提供支持，开放的硬件参考设计和接口标准逐渐变得唾手可得。

例如越来越多的开发者开始通过 Arduino、Raspberry Pi 以及 BeagleBone 快速创建原型。但此类开放式硬件有一个问题，尽管可用于快速创建原型，但如果想将产品投放市场，还需要重新再来一遍，这可能是因为此类硬件产品的许可不允许用于最终产品，或者组件过于廉价不适合用于商用级别的产品。开放式硬件平台也在进化，其中一些已经可以同时用于原型和商用产品中。

开发者需要寻找对业务更为友好的开源许可，要找到以开源方式提供的工业级组件以及一系列工具，随后才能更快速地将有关 IoT 的创意从最初的原型变为可量产的大规模部署。实际上这一过程需要进行的大量集成、

测试，以及验证工作已经预先完成了，就算需要扩展为全球化规模，开发者也不需要付出太多成本。

通过开放式硬件加速 IoT 开发

上述开放式平台可以让开发者借助有限的硬件、无线网络，或低层软件开发经验，在数天而非数月内完成应用程序的开发。若能妥善运用，开源平台和硬件之间的相互通信能力可确保各种连接器和传感器无需额外编写代码便可自动配合使用，这样便可以大幅缩短从构思和原型，再到最终量产过程所要付出的时间和精力。借助工业级的规范，此类下一代平台不仅可用于快速创建原型，而且可以快速实现 IoT 应用程序的工业化生产，因为原型可直接进入量产阶段。

配合使用多个供应商和多个平台，这样的能力为第三方合作关系和 IoT 初创企业提供了大量新机遇。这样的做法为新一代互联应用程序奠定了基础，使得开发者可以无需考虑所用设备直接开发 IoT 应用程序。

更完善的生态系统支持

开源解决方让项目在投资和周期方面更经得起考验，项目完成后多年时间内均可通过各种资源和工具不断对项目进行完善。这样的特性不仅可以保护解决方案开发阶段所付出的时间和投资，也可以通过简化的过程缩短从创新到最终上市所需的时间。

软件方面，如果使用能得到广泛支持的开源软件应用程序框架和开发环境，例如 Linux，将能为开发工作提供极大的帮助。如果使用专有解决方案，只能从原始供应商处获得有关开发框架的支持，而这些供应商的规划可能无法与您的需求保持一致。开源解决方案可以通过更广泛的开发社区为您提供帮助，确保哪怕在多年后您依然能找到各种实用的开发资源。

您在解决方案开发过程中投入的时间和成本也能获得更妥善的保护。

使用开源软件还能获得另一个优势。例如，为数众多的开发者确保软件代码可以经历更严格的审查，这样您的解决方案也能更安全。此外 IoT 应用程序开发者还可以根据具体安全需求对代码进行修改。

没人可以预见 IoT 技术适用的每个应用场景，但基于标准和开源技术的战略有助于促进 IoT 创新，让开发者能够以更快，更简单的方式将更长生命周期的应用程序投放到市场。标准保障了技术的互操作性，开源项目保障了软硬件组件在产品和服务平台演进之后依然可以实现复用。如果不这样做，IoT 领域的创新将依然面临各种阻碍。

现在 Google 制造自己的芯片 Intel 要发疯

作者 CADE METZ 译者 大愚若智

GOOGLE 已经制造出自己的计算机芯片，而事情还远没有结束。

整个互联网上最强大的公司昨天在科技界抛出了几枚重磅炸弹，披露了全新的定制化芯片，借此这家庞大的在线帝国可以更好地经营未来的主营业务：人工智能。

为了制造自己的芯片，Google 在改造了科技界的很多东西之后，又向前迈近一步。为巩固各类在线服务，过去十多年来这家公司为自己的大规模数据中心设计了各种新硬件，包括计算机服务器、网络设备等。随着服务范围和规模达到一个空前的高度，他们需要通过更高效的硬件运行这些服务。多年来，其他很多互联网巨头也曾效仿这种做法，以此为契机全球硬件市场产生了翻天覆地的变化。

Google 在芯片制造方面努力的结果辐射范围已经超越了 Google 帝国本身，甚至让芯片行业的未来受到威胁。

对于这款新芯片，Google 的目标始终未变：空前的高效。为了让 AI

技术跃上一个新高度，他们需要一种能在更低能耗前提下，用更短时间完成更多任务的芯片。但这个芯片所产生的效果已经超越了 Google 帝国本身，甚至让 Intel 和 nViaia 这样的商业化芯片制造商的未来受到威胁，尤其是考虑到 Google 对于未来的愿景，这种情况显得更为紧迫。根据在 Google 帝国的巩固之路上主要负责全球数据中心网络的 Urs Hölzle 所说，新的定制芯片只是万里长征的第一步。

不，Google 不打算将自己的芯片出售给其他公司，他们不会与 Intel 或 nVidia 直接竞争。但考虑到这家公司的数据中心规模之大，Google 目前已经是这些芯片公司最大的潜在客户。与此同时，随着越来越多企业开始使用 Google 提供的云计算服务，他们自行购买的服务器（以及芯片）数量只会越来越少，这等于进一步蚕食了芯片制造商的市场份额。

确实，Google 公布有关新芯片的新闻只是为了宣传自家的云服务业务，让更多企业和开发者选用自家的 AI 引擎，并将其用于自己的应用程序中。随着 Google 开始将 AI 的强大能力出售给其他公司，这实际上是在（以相当高调的方式）宣称他们可以提供运行这种 AI 的最佳硬件，而且绝无仅有的硬件。

Google 对速度的渴求

Google 的新芯片叫做 Tensor Processing Unit（张量处理器），即 TPU。这是因为这种芯片更适合运行 TensorFlow，正是这个软件引擎驱动着 Google 的深度神经网络，硬件和软件组成的网络可以通过分析海量数据学习如何完成特定任务。其他技术巨头通常使用图形处理器，即 GPU 运行自己的深度神经网络，而这 GPU 最初是针对游戏和其他图形密集型应用程序的图像渲染任务设计的。虽然 GPU 在设计上很适合用于运行驱动深度

神经网络所需的计算任务，但 Google 认为如果使用定制芯片可以进一步提高效率。

根据 Google 的介绍，他们结合机器学习的具体需求对 TPU 的规格进行了调整，用更少量事务就可以运行每一步操作。这意味着这样的芯片每秒钟都能执行更多操作。

目前 Google 同时使用 TPU* 和 *GPU 运行自己的神经网络。Hölzle 拒绝详细介绍 Googe 对于这种 TPU 的使用方式细节，不过透露说这种技术可以处理驱动 Android 手机语音识别功能所需的“部分运算量”。同时他还说 Google 即将发布一篇介绍这种芯片好处的论文，并且还会继续设计能够以其他方式处理机器学习任务的新芯片。最终，他们的目标是彻底停止使用 GPU。“已经取得一些进展了，”Hölzle 说：“对机器学习来说，GPU 还是不够专精，毕竟从设计上就不是针对这种用途的。”

nVidia 可不愿意听到这种说法。作为全球最主要的 GPU 经销商，nVidia 正在急迫地将自家业务拓展到 AI 领域。同时 Hölzle 也提出，最新款 nVidia GPU 包含一个专门的机器学习模式。但是很明显，Google 希望这个变化能够进行的快点，再快点。

最智能的芯片

与此同时，其他公司，尤其是 Microsoft 也在芯片之路上进行着探索。场效可编程门阵列（Field-programmable gate array，FPGA）就是一种可以通过重新编程执行特定任务的芯片。Microsoft 已经测试过 FPGA 在机器学习领域的运用，而 Intel 也明确了市场的发展方向，最近收购了一家销售 FPGA 的公司。

一些分析师认为这才是最明智的发展之路。密切关注芯片制造业

务的 Moor Insights and Strategy 事务所总裁兼首席分析师 Patrick Moorhead 认为 FPGA 可以提供更高灵活性。Moorhead 怀疑 Google 新发布的 TPU 是否有些“过犹不及”，并指出制造这样的芯片至少需要六个月，在竞争日趋激烈的市场中，六个月时间已经太长了，并且还要与最大的互联网公司展开竞争。

但 Google 不想要这样的灵活性。对他们来说，速度的重要性超越一切。在被问到为什么 Google 要从零开始制造自己的芯片而不是使用 FPGA 时，Hölzle 说：“只是想实现更高的速度。”

核心业务

Hölzle 还提到 Google 的芯片并不是为了取代 CPU。中央处理器是所有计算机服务器的心脏，这个搜索巨头依然需要使用这种芯片运行数据中心内成千上万的服务器，而 CPU 是 Intel 的主营业务。当然如果 Google 愿意专门为了 AI 打造一款自己的芯片，那么人们不禁好奇他们是否打算更进一步重新设计自己的 CPU。

Hölzle 淡化了这种可能性。“只需要解决尚未解决的问题，”他说。换句话说，CPU 是一种成熟的技术，已经可以按照预期正常工作。但同时他也说，Google 希望芯片市场能够呈现出良性竞争的环境。也就是说，他们希望从多家经销商处购买，而不只是购买 Intel 的产品。毕竟更激烈的竞争对 Google 而言意味着更低的价格。同时 Hölzle 也解释说，Google 正是为了能获得更丰富的选择而与 OpenPower 基金会合作，这家机构的目标是提供任何人都可以使用和修改的芯片设计。

这是个很强大的想法，同时对全球最大的芯片制造商来说也是一个很强大的潜在威胁。根据 IDC 研究公司分析师 Shane Rau 所说，全球服务器

CPU 出货总量中有大约 5% 是 Google 购买的。他说在最近一年内，Google 购买了大约 120 万块芯片，其中大部分来自 Intel。（2012 年，Intel 高管 Diane Bryant 告诉 WIRED 说 Google 从 Intel 处购买的服务器芯片数量远远超过其他五家客户，而其他这些客户全都是销售服务器的公司。）

无论对 CPU 有何计划，Google 还将继续探索更适合机器学习需求的芯片，而我们要等到很多年后才能知道什么可行，什么不可行。毕竟神经网络这个概念本身也在继续进化。“我们的探索从未中断，”他说：“但最终答案是什么我还不知道。”可以肯定的是，随着 Google 继续探索，全世界的芯片制造商都在密切关注。

微软开发团队的 DevOps 实践启示

作者 Thiago Almeida 译者 周小璐

过去几年，微软的工程师团队已经接受了 DevOps 的工作方式，本文讲述我们在这个过程中积累的经验。

纵观整个软件产业，坦白地说，从我们一路的经验来看，DevOps 的实践和方式对于服务和其它产品的交付起到了至关重要的作用。而且，据我们观察：在拥抱 DevOps 过程中，组织的变化和文化导向也是很有意义的。这导致了我们团队的组织结构变化，每个人职责的变化，以及开发、运维和业务文化的变化。

比如，过去在工程实践和工具中，有 Windows、Office 和其它的区别。现在每天有 43000 位来自不同工程师团队的内部用户使用 VSTS（Visual Studio Team Services），并且这个数量在急剧增长，我们非常希望 VSTS 能成为支持工程师团队实践的默认工具。

在本文中，笔者总结了微软工程师团队（尤其是云 & 企业团队和 Bing 团队）现有的演讲，以及内部讨论，希望有助于大家的 DevOps 实践。

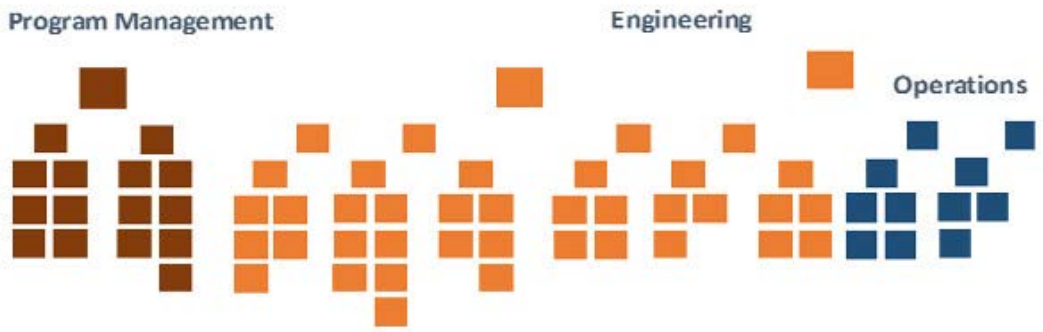


图 1

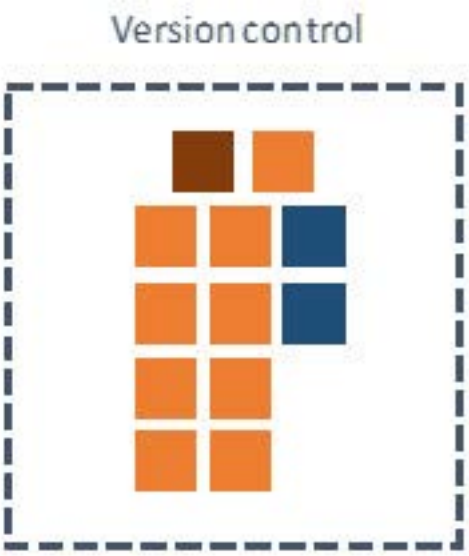
团队组织

过去在工程师团队中有三个角色：项目经理，开发人员和测试人员，从组织和团队的角度来说，开发和测试是完全区分开的。并且，运维团队又是工程师团队之外的一组人。

我们希望减少开发人员和测试人员的交接时间，让他们专注于软件的质量，所以我们将传统的开发人员和测试人员合并为了软件工程师。目前产品功能实现的所有方面都由软件工程师负责，他们还要保证在生产环境的稳定运行。这并不意味着测试被抛弃了，相反的是测试和软件质量成为了每个人的责任。

并且，为了交付最好的服务给用户，我们需要工程师团队和运维团队在从设计到生产环境部署的整个开发周期中，紧密协作。我们首先将运维团队合并到了工程师团队中，运维人员的心态和职责都发生了重大改变，出于这个原因我们称运维团队为售后工程师。

为了交付最好的服务，我们必须将团队统一。写代码的人员和服务维护人员的高度耦合，让我们能够更快地交付功能到生产环境。因此最新的组织形式就像下图所示的样子，开发、测试和运维共同构成了工程师团队。（见图 1）



从那之后我们有了功能团队，整个团队专注于同一个解决方案，功能或产品。在开发部门，这个团队为开发人员和开发团队提供工具，功能团队由 10-12 个人组成，并且是自我管理的，这个团队大概会保持 12-18 个月。目前在开发部门有 4307 人，其中 436 人属于售后服务团队，他们又有 35 个功能团队。以下是从组织角度来看功能团队：一个项目经理，一个工程师 leader，多个软件工程师和售后工程师，售后工程师隶属多个功能团队，但这些功能不会跨产品。（右图）

另一个有趣的变化是团队的座位，功能团队有了特有的办公区域，这里大家可以随意坐，但是这个区域内所有工作相关的对话都要是和每个人相关的。在这里还可以开会，以及长期的谈话和电话。这是开放计划和办公室的完美结合。

团队职责

但是以上所有行为最重要的目的是配合团队职责的变化，从而让软件工程师和售后工程师为用户提供最好的服务。而且，还有 metrics 功能被实现用来帮助评估进度，和鼓励正向的文化转变。比如，测试覆盖和用户 SLA 是团队共同的职责。

软件工程师的职责将不仅是构建和测试，还要保证最终的产品稳定运行。这种职责转变有两方面意义：第一，我们希望功能团队为了能解决遇

到的问题，去试着理解用户；第二，我们需要功能团队和每个工程师对他们交付的产品拥有自主权。功能团队有权控制整个软件流程。

售后工程师们要知道应用架构是更有效率的问题解决方向，如果基础设施架构的改变，能让团队像 IAC（infrastructure as code）或自动化脚本的方式开发和测试，对服务设计和管理都能有正向作用。自动化是微软在软件周期的各个方面中持续追求的关键主题，提高了向用户扩展和交付服务的效率。比如像测试，环境创建，发布管理这些原先是手工的操作都被自动化了。售后工程师为团队带来了无价的技能，尤其是动态部分和失败增多的时候。

下表展示了运维的功能及其职责的变化：

OPERATIONAL CAPABILITY	PRE-DEVOPS TRADITIONAL OPS	NOW DEVOPS
Capacity Management	Ops	DevOps*
Live Site Management	Ops	DevOps*
Monitoring	Ops	DevOps**
Problem Management	Ops	DevOps*
Change Management	Ops	Dev**
Service Design	Dev & Ops	DevOps
Service Management	Ops	DevOps*

* 代表这部分功能已经部分自动化了。

** 代表这部分功能已经大部分或全部自动化了。

值得注意的是变更管理（Change Management）这个功能从运维转向了开发，这是因为新服务和热修复是由一个对等的审核系统自动部署到生产环境的。自动测试和部署，以及功能标记的引入，降低了风险。

这些变化已经被团队很好地吸收了。过去作为微软 BizSpark 项目的

一员，我曾和很多非常有潜力的初创公司共事过。但是最近在和微软内部的功能团队交流的时候，从他们身上我感受到了和那些初创公司一样的驱动力和激情。

这些变化带来了以下好处：

- 提高了成就感；
- 功能团队愿意去理解用户；
- 有了明确的界限来解耦服务；
- 专注于自动化和遥测。

跳跃式部署

从像 VSTS（Visual Studio Team Services）的托管服务，到 OneDrive iOS 版这种移动应用，微软团队已经意识到了 canary 发布（批量部署）带来的好处。在 VSTS 团队中，Canary 发布被称为部署环。团队自动化了构建和测试过程，并自动部署到内部或早期的 feedback 账户或开发者的物理设备中（也叫 dogfooding）。这样能够控制软件的发布，并获得早期的反馈和实验。

VSTS 团队就采用了部署环的方式，服务的更新被分解为 4 个部署环，分布在 Microsoft Azure 不同区域的 12 个扩展单元里。部署是批量的，VSTS 账户在第一个部署环的第一个部署单元中，在其它 3 个部署环将更新推送给全球其他 11 个用户扩展单元之前。第一个部署环中的发布需要经过功能团队 leader 的批准，后续的发布都是自动的。由于团队内部首先获取更新，他们会首先亲身测试：在工作时间，还有合适的工程师负责修正。如果有错误发生，他们希望能最先知道。

大多数被部署的代码，都带有功能标志，来进行另一个层面的发布控

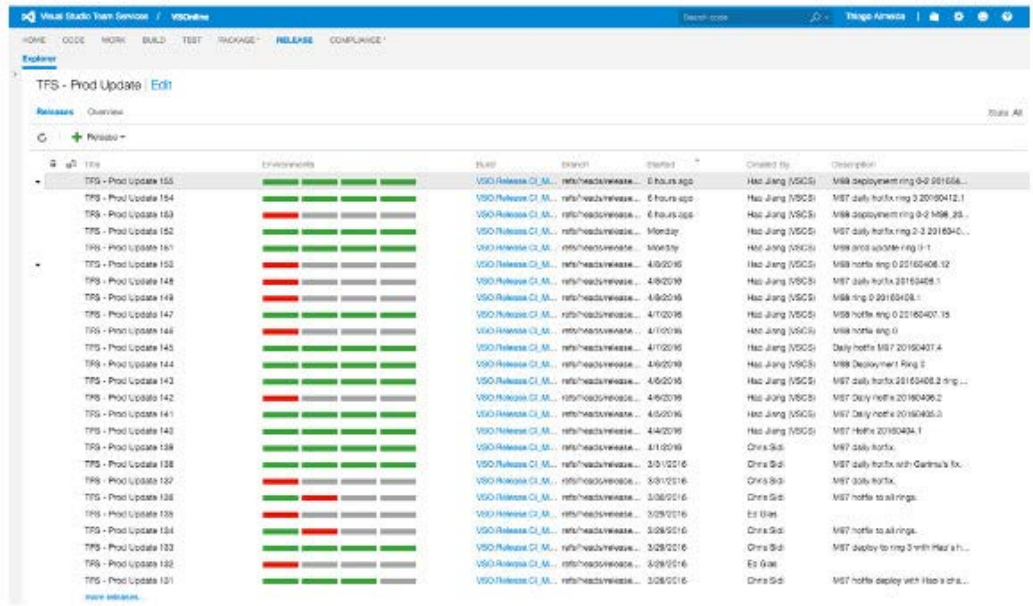


图 2

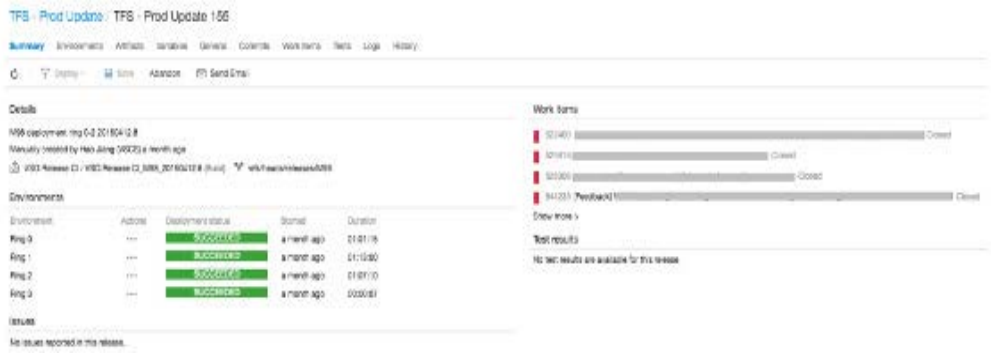


图 3

制。常言道，能自己编译的才是好的编译器。下图中每行都代表一天中热修复的一项条目。在 VSTS 的环境任务是按照逻辑分组的，可能有前后关系，这样并行或串行地执行任务，能保证 VSTS 团队部署环的正常运作。（见图 2）

如图 3 所示：155 号更新被成功发布到了 4 个部署环，同时相关的工作项也被部署到 12 个扩展单元中。

另一个例子是 OneDrive 移动团队。他们使用 VSTS 去自动化编译和测试 iOS 应用，然后 VSTS 会通过一个叫 HockeyApp 的产品，自动推送这些编译到物理设备中。HockeyApp 还能分析所有的冲突数据，这样开发团队

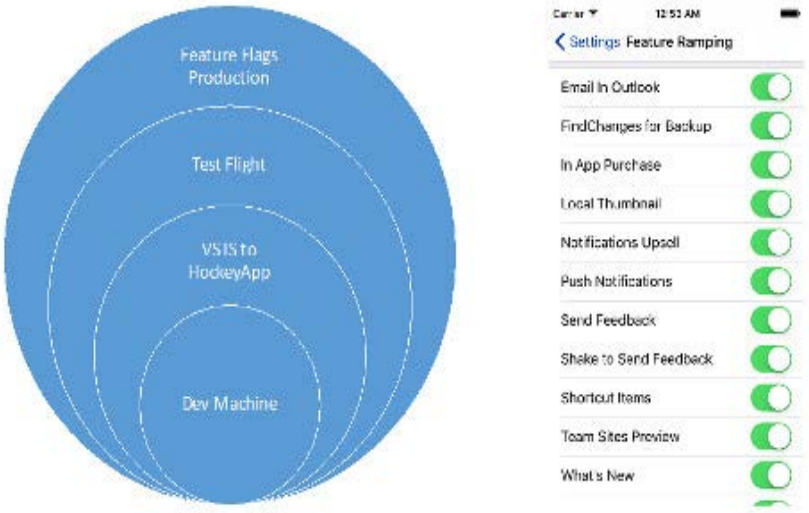


图 4

的成员都能解决问题。他们使用 HockeyApp 发布更新到团队和内部使用者上。

在 OneDrive 团队用 HockeyApp 将发布扩展到开发者和内部用户之外后，这些发布会通过苹果的 TestFlight 给更多的 beta 用户，最终加上功能标志，发布到生产环境。一旦一个功能有了够多的正向反馈和测试，就会最终发布给所有的用户。（见图 4）

这样做带来的好处如下：

- 可以得到早期的反馈，促进了实验；
- 控制了发布流程；
- 内部团队第一个测试，提高了代码质量；
- 有助于减少解决问题的时间。

从用户中学习：直接反馈

有很多分析和遥测技术来支持用户的反馈环路，提高持续交付和假设驱动的工程。

但是我们发现，给用户提供简单和直接的反馈形式，比如在 Microsoft Office 应用中『tell us what you like』和『tell us what you don't like』这样的弹出框，有助于形成一个社区，提高产品质量，拉近用户和开发团队的距离。毫无疑问，等待用户在 tweet 上发布应用或服务的问题，并不是收集直接用户反馈的最好机制。所以，Microsoft Office，Bing 的主页和 Azure Portal 等几个产品上都有精心设计的 feedback 按钮，用户可以直接将反馈发送给功能团队，获得他们的技术支持。

以下是 Microsoft Office 应用上的反馈图标：

很多团队还实现了 UserVoice 功能，或者类似的反馈地点，对反馈进行收集和分组，这些会成为团队的待办项目。User Voice 被用来提供建议和想法，而不是提交 bug，图 5 是 Visual Studio 的 UserVoice 页面。

图 6 是一个移动应用的例子：OneDrive 团队在应用的每个平台上都添加了『contac us』的反馈机制。为了高效地处理这些反馈，他们使用了一个叫做 Parature 的产品。控制台收集了用户数据，集中他们所有的反馈给团队去审核。

直接的用户反馈方式带来了以下好处：

- 提高质量；

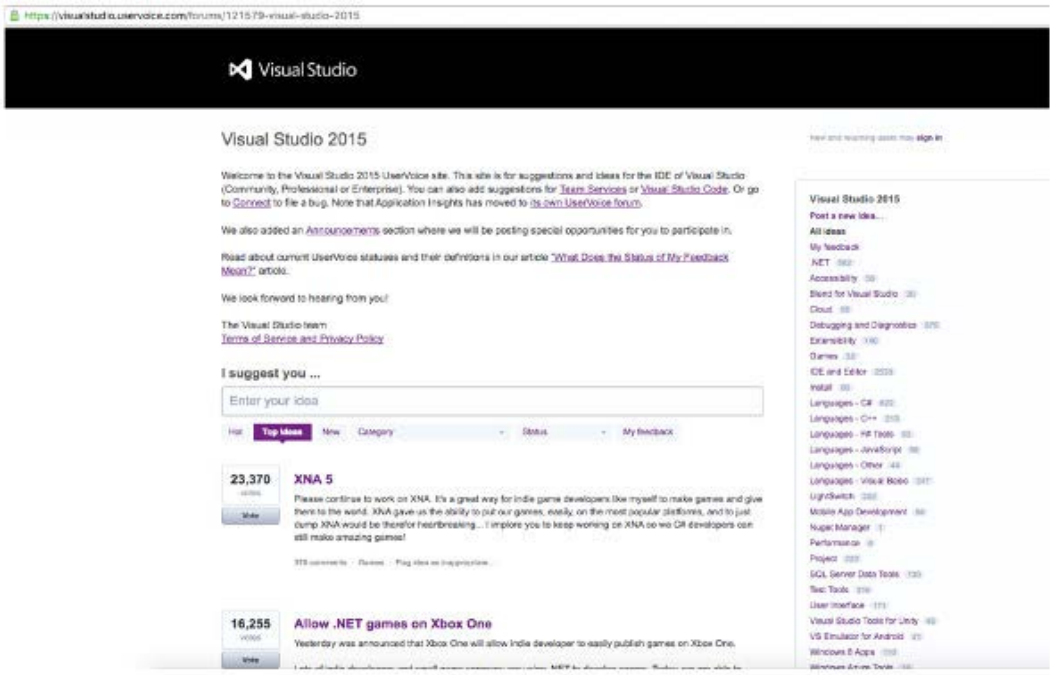


图 5

- 有助于形成社区；
- 功能团队可以更好地理解用户，并获得直接反馈；
- 提高了用户满意度。

Idea Velocity

最近 Idea Velocity 是微软团队都在关注的话题，Idea Velocity 代表实验的速度，代表从一个不成熟的想法到这个功能对用户的影响分析。

个人雇员被鼓励产生新创意，实现它们，并尽快在生产环境中测试。这大部分发生在有成熟的持续交付流程的团队中，它们有稳定的自动测试基础设施，以及各个层面的遥测功能。最重要的指导原则是一个功能的创意来源是多样的。

在 Bing 团队，一旦他们的持续交付和大规模测试就绪后，他们就会集中精力到团队中每个人的亲身测试上。这样个人得到了授权，产品的决策真正出自于用户数据，创造力得到了鼓励。那种认为自动化测试是成功

图 6

关键的说法是非常保守的。

我们将开发团队变为高效的创意漏斗，能尽快地迭代终端用户的想法，从而吸收进来尽可能多的创意。这影响到了团队中的每个人，也真正地拉近了开发者和用户的距离。可以通过一些活动做到这一点，比如增长黑客，孵化器，和 Hack Day。

增长黑客是 VP 级别的，所以是影响组织方向的很好方式。比如，提高开发系统的效率或快速提高用户订单。

BingCubator 是一个论坛，企业可以在其中找到值得投资的创意。这些创意在公开出来融资前，被一个专门的团队（v-team）管理着。

Hack Day 则是模拟了工程师们典型的日常交互，但是让他们暂时放下平时的工作，做一些工作职责之外的事情。

在 Bing 团队，工程师们可以通过工具，在几分钟之内获得外部用户对他们想法的反馈。工程师们提交他们的 mockup 和问题，并选定目标受众。他们的实验就会被发送给上千位用户并获得反馈。微软有自己的众包平台，其中包括了几千个外部用户，这样通常 2 个小时内就能得到反馈，工程师们甚至不用写代码就能验证自己的想法。

你想知道自己关于 Hack Day 的一个想法怎么样？可以做一个快速的原型，形成一个调查问卷，推送给大家，并评估这个原型的可行性。这样开发者们可以获得实时反馈，从而理解他们的创意是否适合被扩展到生产环境中的终端用户。

一旦这个想法被证明了，就会通过持续交付的方式被发布。Idea Velocity 的实现带来了以下好处：

- 解耦了工程师和市场团队；
- 支持早期的反馈和验证；
- 保证了创意的来源是多样的；
- 支持共享标准的文化；
- 减少了HIPPO（Highest Paid Person Override）的决定；
- 鼓励创造性；
- 提高了持续交付，大规模测试和遥测的技术。

总结

这些变化帮助提高了编码效率，质量和产量，从而提高了产品的质量和用户的满意度。但最重要的是我们的开发者们的支持。我们将他们从繁琐的工作中解脱出来，鼓励了最佳的工程实践，从而形成了更好的工程师团队，他们在工作 / 生活的平衡上也更加得心应手。团队的高效意味着他们感到自己所做的无用功减少了，这带来的是他们工作各个方面的提高。

关于作者

Thiago Almeida 在巴西长大，加入微软总部前，在新西兰居住多年。他所在的团队主要是驱动新技术的采用，专注于云计算，开源和 DevOps 实践。


技术 干货
活动大本营
Geekbang 极客邦科技
InfoQ
免费 大会 课程 沙龙

Rust 1.0 发布一周年发展回顾与总结

作者 庄晓立

前言

Rust 1.0 发布刚刚一周年（2015.5~2016.5），这一年来 Rust 又取得了长足的进步。笔者尝试从多个方面总结过去一年来 Rust 领域的重要动作、进度和成就。本文内容丰富，信息量大，总结比较全面。读者从中可以看到：开发者的辛勤努力和 Rust 语言的快速成长，Dropbox 等公司在生产环境中的核心模块应用 Rust，社区成员积极参与社区活动，Rust 在国内的发展状况，等等。

Rust 语言 / 编译器 / 标准库升级

一些零散的升级，像添加 Stable API、局部提升性能、修改某些 BUG 等等，在这里就不提了。我将要说的，都是影响深远的重大升级。当然，还有很多工作未最终完成，要等以后的版本问世。但是前期的研究、讨论、设计等步骤基本走完，剩下的无非就是编码实现、实验性应用、标准化等步骤，只要没有意外，后面的一切都顺理成章。

本文多次提及的 RFCs，后面将有专门章节介绍，此处不展开叙述。

impl specialization (RFC 1210)

这一特性类似 C++ 的模板特化和偏特化。允许为接口或类型定义多个可重叠的 impl 实现，最终由编译器依据上下文自动选择其中一个最具体、最 specific（general 的对立面）的实现。它能帮助程序员更好的优化性能、重用代码，还为将来实现规划已久的 "efficient inheritance" 提供基础支持。

举个简单的例子。Rust 从 1.0 开始就为 “实现了 Display 接口的任意类型 T”

实现了 ToString 接口。这是一个泛型实现，涉及大量类型，覆盖面很广。从代码实现细节上看，用到格式化文本输出（fmt::Write::write_fmt）。

```
001 #[stable(feature = “rust1”, since = “1.0.0”)]
002 impl<T: fmt::Display + ?Sized> ToString for T {
003     #[inline]
004     default fn to_string(&self) -> String {
005         use core::fmt::Write;
006         let mut buf = String::new();
007         let _ = buf.write_fmt(format_args!(“{}”, self));
008         buf.shrink_to_fit();
009         buf
010     }
011 }
```

具体到基础字符串类型 str，它当然也属于上面提及的 “实现了 Display 接口的任意类型 T”，因而自动实现 ToString 接口，拥有 to_string 方法。然而上面的实现代码里的格式化输出功能对于 str 转换至 String 来说是多余的，带来额外的运行时开销，因而导致 "str".to_string() 比 "str".to_owned() 更慢这种很尴尬的局面，持续了一年多时间。

```
001 impl ToString for str {
002     #[inline]
003     fn to_string(&self) -> String {
004         String::from(self)
005     }
006 }
```

Rust 语言支持 impl specialization 特性之后，就可以在标准库里为 str 类型添加一个更具体、更 specific 的 ToString 实现：

impl specialization (RFC 1210) 设计稿自 2015 年 6 月推出第一版，至 2016 年 2 月，期间经过及其广泛而深入的高质量的公开的设计讨论，数易其稿，最终被正式批准采纳。2016 年 3 月，该设计稿被部分实现。impl ToString for str 已经成为现实。

foo?.bar()? (RFC 243)

Rust 现有的错误处理机制（try!+Result）虽然还不错，但是使用有些繁琐，对链式调用不友好。try! 不是语言内置特性，而只是标准库里定义的一个很简单的宏。长期以来，Rust 开发者一直在寻找更好更简洁的错误处理机制，进行了许多探索和讨论。2014 年 9 月 RFC 243 被提出，2015 年 10 月经过重大设计改进，2016 年 2 月被正式采纳，3 月被初步实现，期间经过大量设计讨论。至今 foo?.bar()? 语法已经待在 nightly 版本里被内部测试三个月时间，如果不出意外估计不久就会进入 stable 版本跟大家正式见面。发布之前要做的其中一项工作将是把标准库里面的 try! 全面切换到 ?（已经有自动化工具 untry 可用）。这一特性毫无疑问将改变所有人编写代码的方式。

MIR (RFC 1211)

现在的 Rust 编译器，是从 2010 年开始开发的，是用 2010 年的 Rust 语言编写的。那个时候，Rust 的 0.1 版本都还没出娘胎呢。此后的 5 年时

间里，Rust 语言在进化过程中不断地发生天翻地覆的变动。Rust 编译器正是在这样动荡的环境下逐步迭代完成开发。我们认为，Rust 编译器是由许多个不同版本的 Rust 语言写成的；甚至可以说，Rust 编译器是由许多不同的编程语言写成的（因为每个版本的变化都非常大）。结果就是，在编译器源代码的角落里，难免存在一些陈旧代码和历史遗留问题，并在一定程度上阻碍了编译器的维护和升级。

MIR 计划正是在这样的背景下诞生的。通过在编译器内部引入中层中间表示码（Mid-level Intermediate Representation，介于上层 HIR 和底层 LLVM IR 之间），对编译器代码进行大型重构、改造。这一基础性的改进，对编译器今后的维护、升级，具有深远的影响。许多新特性可在 MIR 的基础上更方便的实现。例如，增量编译、代码优化、更精确的类型检查等等（Non-zeroing drop, Non-lexical lifetimes），都依赖于 MIR 计划的实施。开发者甚至在计划借助 MIR 直接生成 WebAssembly 字节码并已着手行动（WebAssembly：JavaScript 垄断地位的终结者）。

2015 年 7、8 月份完成 MIR 的设计，此后很快进入长达十个月的编码实施阶段。任务量非常大，但进展有条不紊，Github 忠实记录了一切。到 2016 年 5 月份，MIR 代码生成工作基本完成，基于 MIR 的进一步优化和应用，尚在进行中。

alloctors (RFC 1398 1183)

2014 年 4 月提出的 RFC PR 39 未获通过，同年 9 月的 RFC PR 224 也未获通过，直到 2015 年 12 月的 RFC 1398 终于有所斩获，在历经数十次修订后，于 2016 年 4 月修成正果，成为内存分配器标准接口的正式规范，为今后开发者自行实现各种类型的内存分配器扫清了障碍。其背后的动机是，系统

```
tmp/foo2.rs:4:9: 4:18 error: cannot borrow `x` as mutable more than once at a time [E0499]
tmp/foo2.rs:4     let ref mut z = x;
                      ^~~~~~~~~
tmp/foo2.rs:4:9: 4:18 help: run `rustc --explain E0499` to see a detailed explanation
tmp/foo2.rs:3:9: 3:18 note: previous borrow of `x` occurs here; the mutable borrow prevents subsequent moves, borrows, or modification of `x` until the borrow ends
tmp/foo2.rs:3     let ref mut y = x;
                      ^~~~~~~~~
tmp/foo2.rs:5:2: 5:2 note: previous borrow ends here
tmp/foo2.rs:1 fn main() {
tmp/foo2.rs:2     let mut x = [1, 2, 3];
tmp/foo2.rs:3     let ref mut y = x;
tmp/foo2.rs:4     let ref mut z = x;
tmp/foo2.rs:5 }
              ^
```

图 1

```
error: cannot borrow `x` as mutable more than once at a time [--explain E0499]
 --> /Users/jturner/Source/borrowcksnip/rust/tmp/foo2.rs:4:9
3 |>     let ref mut y = x;
  |>         --------- first mutable borrow occurs here
4 |>     let ref mut z = x;
  |>         ^^^^^^^^^ second mutable borrow occurs here
5 |> }
  |> - first borrow ends here
```

图 2

默认提供的内存分配器（例如 jemalloc）不可能适合所有应用场景，第三方内存分配器的引入是必要的。

从整个漫长过程我们看到设计的艰难、设计者的不懈努力，和坚持追求高质量设计的严肃态度。

于此同时，允许变更系统默认内存分配器的设计工作也被提上日程。RFC 1183 在 2015 年 7 月推出设计稿，一个月后获得官方审核批准。8 月份修改编译器，完成编码工作，完整地实现了此 RFC。jemalloc 不再是强制使用的内存分配器，程序员有机会选择使用其他内存分配器作为默认内存分配器。这一功能已被内部测试了将近一年，预计不久之后将会正式发布。

编译错误提示

编译错误提示更加简洁和人性化。

以前与现在的对比见图 1 和图 2。

2016 年 5 月初，此功能已经在 nightly 版本中实现。再经过一段时间的培育，确认其稳定工作之后，将会进入 stable 版本正式面世。（注，上

面的附图是稍早的版本，后面又有所调整和改进。）

其他

增量编译（Incremental compilation）：2015 年 8 月至 11 月完成增量编译的设计文档（RFC 1298）。相关编码实现工作缓慢进展，部分依赖于 MIR。

Macro 2.0：新一代宏 2.0 的规划和设计取得较大进展，nrc 做了大量的前期研究工作，提出了设计稿 RFC PR 1566。接下去的进展有待观察。

impl Trait：取得了许多探索和研究成果（1 2 3），RFC PR 1522 得到了比较广泛的认同，很快会有下一步的动作。完成之后将改善“函数返回迭代器”的现状（struct 扎堆）。

单指令多数据（SIMD）：Rust 对 SIMD 硬件加速支持取得初步进展，huonw 做了大量前期工作，提出 RFC 1199(2015 年 9 月获得批准），提交基础实现和扩展库。属于实验性支持阶段，已在 regex 库中得到应用（性能提升十分明显）。

Rust 周边开发工具升级

Cargo

cargo install：只需一个很简单的命令（cargo install xxx）就能编译安装可执行程序到本地 .cargo/bin 目录（此目录一般都在 PATH 环境变量内）。设计和实现均已完成。像 clippy, rustfmt, cargo-edit 等都能通过它安装，十分方便。

cargo workspace：让 Rust 项目的源代码目录结构和布局更加灵活。设计完成但尚未实现（在本文创作后期已经实现）。

cargo concurrently：允许多个 Cargo 实例并发执行。已经初步完成。

cargo namespacing：官方开发者始终持有主动排斥的不作为态度，因

而一年来没有任何进展。跟 Go 语言泛型面临的悲催处境惊人的相似。

clippy

clippy 得到了许多人发自内心的喜爱。它会分析审查你的源代码，并提出许多贴心的改进意见。它像良师益友，安静地坐在你身边，面对面指导你编写代码。经常使用它，有助于改善代码质量、提升开发者的编程经验和水平，无论新手老手都能从中受益。

Thanks rust-clippy. We do love the great project.

安装方法：cargo install clippy；使用方法：cargo clippy。

rustfmt

格式化 Rust 源代码的工具，已经基本成熟、稳定可用，许多官方库用它格式化代码。不过它存在的意义相比 clippy 弱暴了。rustfmt 只是辅助修整一些皮面而已，对语意几乎一窍不通，我想不通某些哥们经常对这类工具推崇备至。安装方法：cargo install rustfmt；使用方法：cargo fmt 或 rustfmt。

rustup

Rust 安装和更新工具，可以方便的切换 stable 和 nightly 版本，可以下载和使用各种平台交叉编译工具链。前身是 Shell 脚本，现在已经是纯 Rust 开发 (rustup.rs)。开发者首选的 Rust 安装方式。这里还要顺便提及 brson 为方便各操作系统打包 Rust 所付出的努力。

rustbuild

专为 Rust 编译器和标准库定制的基于 Cargo 的编译工具，用于取代之前基于 Makefile 的编译系统（已经过于庞大和复杂难以维护）。已经做了大量前期工作。

Editors & IDEs

各主流代码编辑器 /IDE 都对 Rust 提供或多或少的某种程度的支持。

语法高亮着色、代码自动完成等基本功能都已实现。RFC 1317（讨论）还进行了深入的思考和规划，未来 Rust 编译器和其他工具将主动对 IDE 提供更友好的支持（但是目前还没有看到具体的动作）。

Visual Studio Code (VSCode) + RustyCode

- Atom + Tokamak
- Sublime Text 3 + RustAutoComplete
- IntelliJ IDEA + intellij-rust
- Vim + rust.vim + YouCompleteMe
- Emacs + rust-mode

详情可参考 Are we IDE yet，或参见 Reddit 网友讨论帖：Show me your programming environment。官方网站也有总结。

以上几乎所有编辑器 &IDE 及其插件的背后都离不开大功臣 Racer：Rust“代码提示 & 自动完成”工具。

我现在感觉用 VSCode v1.2 比较顺手，比 Atom v1.8 反应快还更稳定。作为通用的代码编辑器，我原本对 Atom 寄予厚望，但现在来看 VSCode 有后来居上的趋势。

Rust 第三方应用升级

Dropbox Magic-Pocket

2016 年 3 月份，Dropbox 公司脱离 Amazon 云存储平台的战略规划和实施细节浮出水面（低调做事之后高调发布，这风格我喜欢）。国际新闻报道标题是：“The Epic Story of Dropbox’s Exodus From the Amazon Cloud Empire”（英文讨论）。国内新闻报道标题是：“Dropbox 用 Rust 取代 Go 精简内存占用”（中文讨论）。Dropbox 这一计划的核心是自己开发云存储系统，并逐步过渡。一开始是用 Golang 语言开发，后来上线后发

现系统占用内存太厉害，于是改用 Rust 语言开发该系统的核心模块 Magic Pocket。对外报道时，新系统已经上线一段时间，运行状况良好。

Redox OS

Redox OS 是真撸啊。想搞另一个 Linux 出来。

用 Rust 语言开发的 Redox 操作系统，2015 年 9 月一经面世就拥有图形用户界面，现已实现文件系统 (ZFS WIP)、网络系统、多线程、文本编辑器等等。微内核设计，硬件驱动运行在用户空间。兼容大多数 Linux 系统调用和常见的 Unix 命令。

它可在 Linux、Windows、OS X 上编译，可在真实硬件上 (Panasonic TOUGHBOOK CF-18、IBM Thinkpad T-420、ASUS eeePC 900) 启动并运行 GUI 应用和 Console 应用。一个团队还在不断的开发完善 Redox 系统。开发活跃，进展很快，文档较全。

个人或小团队开发出来的绝大多数所谓的操作系统，都是习作、玩具。那些借助 BootLoader 启动起来并在屏幕上输出文字的所谓内核，除了作为新手入门练习之外没有任何应用价值。Redox 显然不属于此类，Redox 有更高的追求。Redox 的作者也不是新手，而是资深的 OS 开发专家。

- Announce Redox
- Redox is Serious
- Redox vs Linux in 10 years
- Rust's Redox OS could show Linux a few new tricks

Redox OS 固然牛逼，可它也不是一个人在战斗，用 Rust 开发的操作系统有好几个呢。

TiKV 分布式存储引擎

PingCAP 公司推出的 TiDB 是开源的分布式数据库，参考 Google F1/

Spanner 实现了水平伸缩，一致性的分布式事务，多副本同步复制等重要 NewSQL 特性。结合了 RDBMS 和 NoSQL 的优点，同时完全兼容 MySQL。前期用 Go 语言实现了 SQL Layer，后来（2015 年底）考虑到“Go 的 GC 和 Runtime 对底层存储非常不友好”，再加上对运行效率的极致要求，决定采用 Rust 语言开发 TiDB 的核心存储模块 TiKV。2016 年 1 月份至今，TiKV 项目的开发十分活跃，至少有 4 位软件工程师全职参与开发。4 月份刚刚开源。

Diesel ORM

作者 Sean Griffin 是 Ruby Rails ActiveRecord 的现任活跃维护者，是 ORM 领域的专家。他在 2015 年 9 月启动 Diesel 项目。来自 ActiveRecord 的优秀设计、经验和教训，有助于他设计实现这个全新的 ORM 项目。Diesel 项目的设计充分发挥 Rust 类型安全的优势，还特别注重性能和可扩展性，文档也不错。是颇受关注、值得期待的项目。

在生产环境中使用 Rust

Rust 官方网站专门有一个页面，Friends of Rust（2016 年 4 月底上线，持续更新），列出了在生产环境中使用 Rust 语言的组织和项目。

其中 MaidSafe 是有野心的项目，这一年来一直持续开发。Servo 也是；而且会有越来越多的 Rust 开发的 Servo 组件迁入 Firefox 浏览器。

其他潜力股项目

- https://github.com/kbknapp/clap-rs
- https://github.com/sebcrozet/ncollide
- https://github.com/google/xi-editor
- https://github.com/nikomatsakis/lalrpop

I would like to nominate lalrpop. It's an LR(1) parser generator, where the syntax is written using a nice, quite Rust like language, and it's compiled into Rust code as a part of the build process. There's also the option to use your own token type and tokenizer, which is very nice for more advanced

projects. (ogeon)

- https://github.com/tailhook/vagga
- https://github.com/das-labor/panopticon
- https://github.com/aturon/crossbeam

Libraries like crossbeam are probably more appropriate for the std. Lock-free containers(data structures). (LilianMoraru)

- https://github.com/nikomatsakis/rayon

Ok, if only 1 crate, I nominate rayon.

This crate is so good that I think that most applications should have a dependency on it.

Very easy to plug some concurrency into your application, and that it does very well. (LilianMoraru)

- https://github.com/Geal/nom
- https://github.com/carllerche/eventual

I would like to nominate eventual. It's a library that provides Future and Stream like abstractions and has a very easy to use interface. (maximih)

注：xi-editor 项目虽然挂在 Google 名下，却不是 Google 公司官方项目；hat-backup 项目的情况也类似。说明 Google 公司至少有两位作者在持续或深度使用 Rust 语言。以后在 Google 公司内部安利 Rust 就靠这哥俩了……

重量级 PR（Pull Request）

- Regex #164: Add a lazy DFA （性能颇具竞争力）
- Url #176: Version 1.0, rewrite the data structure and API
- Hyper #778: Async IO
- Mio #239: Preliminary Windows TCP/UDP support

- Redox #552: Big IO change
- Rust #32756: Overhaul borrowck error messages and compiler error formatting generally
- Gtk-rs #221: Object reform

Rust 社区生态系统升级

Crates.io 中心仓库持续发展

crates.io 下载量 Top 10

- libc 共190万次下载
- winapi 共120万次下载
- rustc-serialize 共109万次下载
- rand 共101万次下载
- winapi-build 共99万次下载
- bitflags 共98万次下载
- log 共86万次下载
- kernel32-sys 共80万次下载
- gcc 共72万次下载
- time 共68万次下载

说明它们处于依赖链的顶层或接近顶层，大量项目直接或间接地依赖之。任意项目每次全新的编译都可能增加其下载次数，自动化的集成测试也贡献了许多下载量。

通过 Cargo 和 crates.io 网站把大大小小的第三方库聚合在一起，形成健康的生态系统，让项目依赖管理变得对开发者透明。

线上线下同性交友网络蓬勃发展

Github 号称全球最大的线上同性交友网，名副其实。单单一个 Rust 仓库，就汇集了 5.3 万 commits，1.7 万 issues，1.6 万 PRs，1.6 万 stars，3 千 forks，1 千 contributors。还有 RFCs 仓库几百个精华的设

计文档和精彩的讨论细节。

大概从 2015 年 8 月起，在 reddit.com/r/rust 论坛上，每周都会发布两个置顶的新帖：

- “兄弟们这周都忙啥了？”（"What's everyone working on this week?"）后面的回复中网友纷纷踊跃发言说自己本周在用Rust写什么代码或做什么项目。
- “哥们有事您说话！”（"Hey Rustaceans! Got an easy question? Ask here!"）下面的回复有问有答，提出并解决各个技术问题。

这情景很和谐。真的就像一帮有情有义的光棍汉哥们围着电脑坐在一起认真的闲聊，聊工作，聊技术。虽然彼此之间隔着网络，心却是在一起的，相互有信任感和亲近感。

他们不把自己喜爱的语言当神供着，该吐槽的时候比谁都积极。在诸如 "Rust 哪些地方最垃圾" "为何不在工作中使用 Rust" 这类问题下面你会看到几百条回复。

线上火热，线下也不闲着。2016 年 1 月份，全球 Rust 开发者共举办了 13 次有据可查的同性聚会；2 月份 15 次；3 月 14 次，4 月份 14 次；5 月份（刚过半）16 次。活该没有女朋友，哼！ （此数据为粗略统计，截止到 2016 年 5 月中旬。）

以上数据部分来自 https://github.com/rust-lang/rust 和 https://this-week-in-rust.org/ 。

Rust 价值观输出升级

以下列举的情况，有些可以证实是 Rust 输出了价值观，有些则不能证实。然而即使不能证实，至少也说明对方持有和 Rust 相类似的价值观，这

也是我们喜闻乐见的。

简短的类型名称被 WebAssembly 采纳

由 Mozilla, Microsoft, Google 等几大浏览器巨头联合发起的 WebAssembly 项目（JS 垄断地位的终结者），其 value types 从 2015 年 9 月开始直接使用跟 Rust 基本类型完全一致的命名：i32, i64, f32, f64。但变更记录并未提及是否借鉴于 Rust。此前他们的 value types 命名曾几经变更。或许只是一个巧合也未可知。

另外，WebAssembly 的 block 也是有值的，block 的值就是 block 内最后一条 expression 的值。同样跟 Rust 不谋而合。

动力火车版本发布机制被 Github Atom 和 Ethcore Parity 采纳

Rust 的 Release channels 机制原本是借鉴于 Chrome，进而又影响了 Github Atom 和 Ethcore Parity 项目。Atom 和 Parity 均声明受 Rust 影响而采用此发布机制。

这一机制可以表述为三列火车（nightly, beta, stable）在各自轨道上并驾齐驱。nightly 火车，每天都往上装货（commit code），内容是最新的，但是可能不太稳定；beta 火车，每隔 6 周从 nightly 火车上卸下一部分经过时间沉淀的货物装到自己车上（merge code），内容比较新，稳定性比较好；stable 火车，每隔 6 周从 beta 火车上卸下一部分经过时间沉淀的货物，经过列车长人工过滤，确认检验无误之后装到自己车上，测试时间最长，测试最完整，稳定性最好，内容相对滞后。一个新特性从提交代码进入仓库到进入 Stable 版本，至少要经历 nightly->beta、beta->stable 两个周期，约 3 个月时间，才能跟最终用户相见；期间经历各种测试，一旦发现问题就会延迟进入 Stable 版本。但无论如何，最终用户总会每 6 周得到一次 Stable 版本升级。

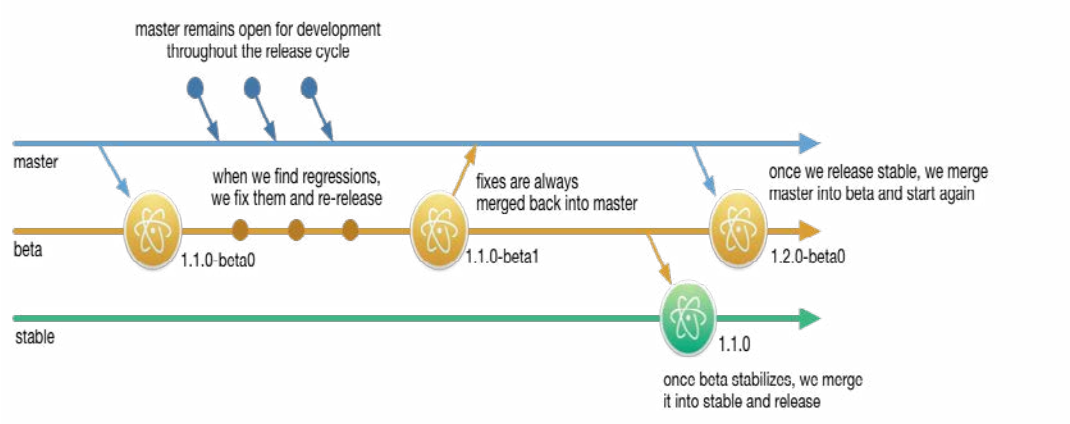


图 3

Atom 提供的这幅图十分形象的揭示了这一机制工作原理。（见图 3）

这种机制的好处是，在保证产品稳定性的前提下提供较频繁且平滑的升级体验。让用户没有心理负担地升级，乐于使用最新稳定版本，而不是像 Java 那样总想着憋 N 年搞一个大新闻出来，反而让某些用户有升级恐惧症。喜欢尝鲜的Rust用户，可以选用每日更新的nightly版本(或beta版本)。两边都不耽误。

RFCs

Rust 早在 2014 年就开始逐步实施 RFC 机制。要求对代码做出重大改动或提议增加重要特性之前，发起者需要首先写出设计文档（即所谓“请求讨论稿” - Request For Comments），经过公开讨论逐步修正完善，并得到核心专家组批准之后，才能进入下一步编写代码实现功能阶段，最后还有一个标准化步骤，直至进入正式发行版 (stable)。

这一机制的好处是：

- 强调设计在编码之前
- 强调公开的设计、公开的讨论、广泛地征求意见
- 强调核心专家组对设计方向和质量的把控
- 有利于积累专业的技术设计文档

- 避免某些模块的设计仅有个别人理解
- 提高代码的可审查性和可维护性
- 保证项目的高质量和发展方向
- 配合Feature-gate确保实验性项目在实验期间不流出实验室

这对开源软件的健康发展是至关重要的。别的项目也开始认可这一点，并参考实施，例如：

- https://github.com/emberjs/rfcs
- https://github.com/maidsafe/rfcs
- https://github.com/intermezzOS/rfcs
- https://github.com/rebuild-lang/rfcs
- https://github.com/fsharp/FSharpLangDesign

注：目前不能证实微软 F# 项目的 RFC 机制源于 Rust。运作机制虽有所不同，但设计在前、公开讨论、官方审核等核心内容是一致的；RFC 文本结构相似度很高。多讲一句，说起来 F# 和 Rust 还是亲戚呢，语言设计层面都跟 OCaml 语言有很深的渊源，Rust 最初版本的编译器还是用 OCaml 开发的（当然现在早就换成 Rust 开发了）。

本周更新公告 (This Week In XX)

“This Week In Rust”每周推出一期，总结介绍上一周 Rust 发生的重要事件，如功能增加或变动，新的设计文档 (RFC) 得到批准，谁发表了重要文章，谁举行了聚会等等。从 2013 年 6 月 Rust 推出第一期“This Week In Rust”开始，三年来共发布 130 期（其中在 1.0 之后发布约 50 期）。可以当作新闻报纸摘要来读，用户花少量的时间就能了解 Rust 重要开发动态。

后来有些项目开始学习这种公告形式：

- This week in Servo
- This week in Redox
- This week in Ruma
- This week in Rust Docs
- This week in intermezzOS
- This week in TiDB
- This week in Parity

这种事情最难得的就是持之以恒。Rust 出到第 130 期了，Servo 出到第 62 期了，Redox 出到第 14 期了，加油吧。（注：因本文创作时间跨度较大，以上数据可能稍有过时。）

别的项目没有（各种形式的）每周（或每月）更新公告，其中一个理由可以理解为：开发活跃度低，每周更新的内容少，不好意思写出来；或者被关注度低，写出来也没多少人看，所以干脆不写了；再或者开发透明度低，不能写出来给人看。（唉，人艰不拆，说出实话就要得罪人！）

支援兄弟语言项目

本着共建和谐开源大家庭原则，Rust 社区充分发扬共产主义精神，积极参与其他社区活动，为各兄弟语言项目送温暖、献爱心，共享发展理念，致力于实现共同富裕。Rust 在安全和性能方面优势非常突出，尤其适合为其他语言编写 Native 扩展库。下面提到的项目多是名家作品。

Lua

有过 Lua 本地模块开发经历（htmlua iedom）的我，第一次看到 hlua 就有眼前一亮的感觉，眼珠子都快掉下来了，惊叹居然可以这样写 Lua 的本地扩展模块，完全突破了 Lua C API 的死板接口：

hlua 的作者是何方大神？这里先卖个关子。后面他还会出现。

Ruby

```
#![feature(phase)]
#[!plugin(rust-hl-lua-modules)]
#[export_lua_module]
pub mod mylib {          // <-- must be the name of the Lua module
    static PI: f32 = 3.141592;
    fn function1(a: int, b: int) -> int {
        a + b
    }
    fn function2(a: int) -> int {
        a + 5
    }
    #[lua_module_init]
    fn init() {
        println!("module initialized!")
    }
}
```

借助 helix 项目，让 Rust 语言编写 Ruby 本地库变得很简单，消除了大量胶水代码。

Helix 的作者是 Rust 核心开发者 Yehuda Katz 和 Ruby on Rails 核心开发者 Godfrey Chan。这个阵容够强大吧？

```
declare_types! {
    class Console {
        def log(self, string: &str) {
            println!("LOG: {}", string);
        }
    }
}

$ irb
>> require "console/native"
>> Console.new.log("I'm in your rust")
LOG: I'm in your Rust
```

Node.js

通过 neon 项目，用 Rust 语言编写 Node.js 本地扩展包，既简化了代码编写，又简化了编译过程。看看它的 demo，让人印象深刻：静态编译 + 并发计算，性能远超 JavaScript，强到没朋友。简单贴一段代码。

Neno 的核心开发者 Dave Herman，是 Mozilla 公司员工，ASM.js 的作者，牛逼的不要不要的。

Golang

```
001 dfn search(call: Call) -> JsResult<JsInteger> {
002     let scope = call.scope;
003     let buffer: Handle<JsBuffer> = try!(try!(call.arguments.
     require(scope, 0)).check::<JsBuffer>());
004     let string: Handle<JsString> = try!(try!(call.arguments.
     require(scope, 1)).check::<JsString>());
005     let search = &string.data()[..];
006     let total = vm::lock(buffer, |data| {
007         let corpus = data.as_str().unwrap();
008         wc_parallel(&lines(corpus), search)
009     });
010     Ok(JsInteger::new(scope, total))
011 }
012
013 register_module!(m, {
014     m.export("search", search)
015 });
```

通过 rure 项目，Rust 给 Go 语言送去性能更强的正则表达式库，让 Go 社区的同学们提前用上高大上的 lazy DFA 特性。

Rure 的作者 Andrew Gallant，是 Rust 开发组成员，Rust Regex 和 Docopt 库的主要作者。

Python

rust-cpython 使得采用 Rust 语言编写 Python 扩展库和调用 Python 代码成为现实。

```
001 #![crate_type = "dylib"]
002 #[macro_use] extern crate cpython;
003 use cpython::{Python, PyResult, PyObject};
004
005 py_module_initializer!(example, initexample, PyInit_example, |py, m|
     {
006     try!(m.add(py, "__doc__", "Module documentation string"));
007     try!(m.add(py, "run", py_fn!(py, run())));
008     Ok(())
009 });
010
011 fn run(py: Python) -> PyResult<PyObject> {
012     println!("Rust says: Hello Python!");
013     Ok(py.None())
014 }
```

Erlang and Elixir

```
#![feature(plugin)]
#![plugin(rustler_codegen)]

#[macro_use]
extern crate rustler;
use rustler::{ NifEnv, NifTerm, NifResult, NifEncoder };

rustler_export_nifs!(
    "Elixir.TestNifModule",
    [("add", 2, add)],
    None
);

fn add<'a>(env: &'a NifEnv, args: &Vec<NifTerm>) ->
  NifResult<NifTerm<'a>> {
    let num1: i64 = try!(args[0].decode());
    let num2: i64 = try!(args[1].decode());
    Ok((num1 + num2).encode(env))
}
```

Rustler 使得 Rust 可以轻松编写 Erlang NIF（本地实现函数库）。

C and others

C 同学也未被遗忘，Rust 双手奉上 Regex C API，并希望 C 同学向各编程语言转达来自 Rust 语言的问候。Servo 也有提供 C API。

Rust 语言内置支持与 C 语言的互相调用（FFI）。其他语言以 C 为中介可间接地实现与 Rust 交互。对于 Rust 调用 C 库的情况，rust-bindgen 很给力，可自动生成 Rust 端调用接口。新特性 cdylib 使得 Rust 编译出的 DLL/SO 文件尺寸更加小巧，更方便被其他语言嵌入、调用。Rust 当然也支持引入和输出静态库 lib 文件。

Vulkan

Vulkan 规范今年 2 月份刚发布，Rust 迅速跟进，很快就有了安全而高效的 vulkano 第三方封装库。

vulkano 的作者 tomaka，同时也是大名鼎鼎的 glium 和前面提到的 hlua 的作者。

最受群众喜爱的编程语言

在国际知名网站 StackOverlow 主办的 2016 年开发者调查报告中，Rust 荣获最受群众喜爱的编程语言大奖。群众的眼光是雪亮的。

Rust 在国内的发展情况：

- QQ和微信群/讨论区Rust中文社区。
- 国内多位开发者合力完成的原创性Rust教材Rust Primer（诞生记）。
- 国内开发者Elton主导并持续开发的协程项目coio-rs。
- 2015年6月起，号称全球最大IT社区的CSDN网站先后多次专访国内活跃Rust开发者（庄晓立(Liigo) 钟宇腾(Elton/tonytoo) 唐刚(daogangtang) 王川(Fantix) 冯耀明(loveisasea)），并推出Rust学习路线和Rust知识库，持续关注和推动Rust发展。
- 2016年3月，PingCAP首席架构师唐刘发表Rust语言入门、关键技术与实战经验（今日头条链接）。
- 2016年4月，本文作者庄晓立(Liigo)在 QCon北京2016|全球软件开发大会 上做专题演讲 《Rust编程语言的核心优势和核心竞争力》（演讲稿PDF）。这应该是Rust语言第一次正式出现在国内顶级软件大会现场。
- 上文提到的TiKV开源项目的开发商PingCAP是中国公司，总部在北京市海淀区。是目前唯一一家出现在Rust官网Friends of Rust页面的国内公司。
- pencil的作者是北京人，sapper的作者是四川人。
- 2016年5月，在北京和成都举办了庆祝Rust 1.0发布一周年的线

下同性聚会活动。

后语

Rust 语言还是成长中的孩子。它有已经成熟的一面，且持续保持稳定，经历过至少一年的时间考验，证明它初步拥有独当一面的能力。它还有欠缺的一面，许多地方有待完善和拓展，它正视自身存在的问题，从多方面努力争取拿出最好的解决方案。过去一年的主题可以总结为：巩固已经稳定成熟的领域，发展全新的领域或有欠缺的领域。Rust 开发者们卓有成效的工作，使得 Rust 取得了可喜的进步，让我们对 Rust 的未来充满信心。Rust 社区高度活跃的现状昭示着我们不是一个人在战斗。Dropbox 等公司的加入使得队伍不断扩大。继续努力吧，务实的 Rust 编程语言，我们期待下一年取得更大的进展。

微服务与服务团队在 Amazon 的发展

作者 Jan Stenberg，译者 邵思华

在早先举办的 I Love APIs 2015 大会上的一场演讲中，Chris Munns 讲述了 Amazon 如何创建企业级的微服务架构的话题。微服务模式改变了我们创建应用的方式，而要成功地创建与运行这些微服务，团队的结构将起到至关重要的作用。

Munns 是 Amazon 的 DevOps 部门的业务开发经理，他在演讲中引用了维基百科上微服务的定义，但同时也列举了微服务的 4 条使用上的限制：

- 单一目的。
- 仅通过API进行连接。
- 通过HTTPS协议进行连接。
- 微服务之间大体以黑盒的方式展现。

Munns 将微服务与 SOA 进行了比较，列举了以下这些差异点。

微服务	SOA
使用大量小组件	存在各别较复杂的组件
业务逻辑存在于单独的服务领域中	业务逻辑可以跨多个领域存在
使用简单的连接协议，例如 HTTP 与 XML 或 JSON	企业服务产总线（ESB）充当了服务之间的层的角色
通过 SDK 与客户端连接 API	使用中间件

描述团队的规模有一个著名的术语，即刚好能吃完两只披萨的团队。在 Amazon，这样的团队被称为服务团队，他们对于创建过程具有完全的

自主权，包括产品的计划、开发工作、运维以及客户支持。他们具备完全的自主权及责任性，同时也负责每日的运维和维护工作。换句话说，谁创建的服务，就由谁负责运行。这意味着质量保证（QA）人员以及运维人员都隶属于服务团队之中。但 Munns 也提到，承担这一角色的部分员工也有可能由整个组织共享。

对于团队来说，这样的文化意味着很高的自由度，但这些团队将通过以下途径得到授权并保证实施的高标准：

- 全面的培训。
- 由具有20年以上开发经验的员工全面定义各种模式与实践。
- 在业务与技术两方面定期进行衡量指标审查。
- 由内部的专家分享关于新工具、服务与技术的知识。

Munn 对于小型团队与微服务在 Amazon 的发展进行了深入的观察，以了解其重点所在。对于其他打算按照相同方式发展的组织，Munn 提出了一些建议：

- 文化 —— 这里要强调一点，自主权与责任是不可分离的，规模越大的团队，其运作速度相对于小型团队将有所下降。团队要坚持卓越产品的标准，但并非坚守做事的方式一成不变。
- 实践 —— Munn提到了持续集成（CI）与持续交付（CD），以及简化运维任务的重要性。
- 工具 —— 这些工具将用于之前所提到的实践、基础设施的管理、指标的设立和监控，以及交流和协作。

Munns 最后强调了为服务和客户建立起一种模式的重要性，这将使组织避免重复发明一些相同的基础部件，将精力浪费在通信、授权、防止滥用和服务发现等任务上。他还阐述了构建、托管、服务的指标对于观察基础设施是否按预期运行、SLA 是否得到满足等问题的重要性。

Geekbang. 极客邦科技 | EGO EXTRA GEEKS' ORGANIZATION NETWORKS

技聚·深圳
ArchSummit
大会精华抢先看

EGO深圳广州七月分会活动

超级极客邦EGO
Extra Geeks' Organization

微服务架构：Kafka 的崛起

作者 Braedon Vickers 译者 王庆

正如我们在以前的博客中提到的，比如说我们的 Docker 系列博客，我们正处于把系统迁移到微服务新世界的过程中。

促成这次架构改造的一项关键的技术就是 Apache Kafka 消息队列。它不仅成为了我们基础架构的关键组成部分，还为我们正在创建的系统架构提供了依据。

Kafka 概览

虽然这篇文章的目的不是在宣扬 Kafka 比其他消息队列系统更优秀，但是本文讨论的某些部分是专门针对它的。对于外行来说，Kafka 是一个开源的分布式消息队列系统。它最初是由 LinkedIn 研发，现在由 Apache

软件基金会维护。和其他的消息队列系统一样，你可以给它发送消息，同时也可以读取消息。用 Kafka 的说法就是“生产者”发送消息，“消费者”接收它们。

Kafka 的独特性在于它同时提供简单的文件系统和桥接这两种功能。一个 Kafka 的代理器（broker）最基本的任务就是尽可能快地将消息写到磁盘上的日志中，并从中读取消息。消息队列中的消息在持久化之后就不会丢失了，这是整个项目的核心所在。

队列（Queue）在 Kafka 中被称为“主题”（topic），同一个主题共享 1 个或多个“分区”（partition）。每条消息都有一个“偏移”（offset）——一个代表它所在分区位置的偏移量。这使得消费者可以记录它们当前读到的位置，并向代理（broker）请求读取接下来一条（或多条）消息。多个消费者可以同时读取同一个分区的数据，每个消费者从某个位置上读取数据都是独立于其他消费者的。它们甚至可以在整个分区内随意跳跃式前进或后退。

多个 Kafka 代理组合到一起成为一个集群。分区是在分散在整个集群中的，这样就提供了可扩展性。它们也可以复制到集群中的多个节点上，实现了高可用性。综合复制与分区持久化特点，进而达到高可靠性。

想了解更多的细节，请阅读这里。

构建微服务的系统架构

为了帮助我们理解 Kafka 的用途和影响力，想象一个通过 REST API 从外部接收数据的系统，进而用某种方式进行变换，最后将结果保存到数据库中。

我们想要把这个系统升级为微服务架构，所以首先把这个系统切分为

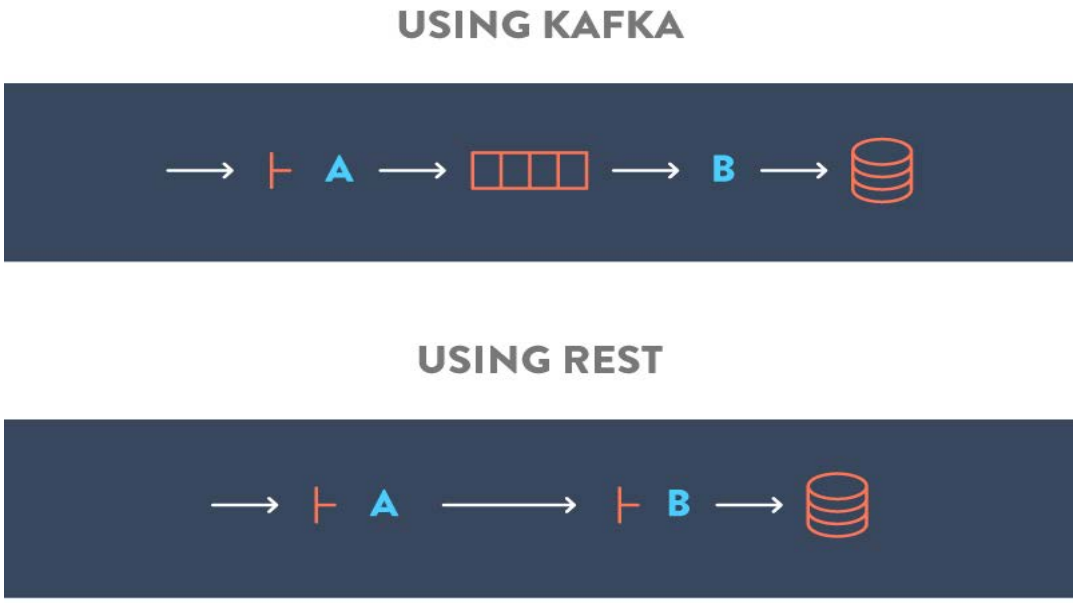


图 1

两个服务 - 一个用来提供外部的 REST 接口（Alpha 服务），另外一个用来做数据变换（Beta 服务）。简单起见，Beta 服务同时会负责存储变换后的数据。

由一对微服务组成的系统会比提供整体服务（往往是糟糕的）的系统要好一点，所以我们定义一个接口用于从 Alpha 服务将数据发送到 Beta 服务，用来减少耦合。和使用 REST 接口实现的系统相比，让我们看看使用 Kafka 实现这个接口将会如何影响该系统的设计和运行。（见图 1）

系统的可用性和性能

当数据被提交到 Alpha 的服务，它在响应客户端之前需要确保数据安全地存储在某个地方，或者已经失败 - 客户端需要知道，因为如果出现了错误它可以重新发送（或用一些其他的方式进行恢复），

使用 REST 接口，Alpha 服务在响应客户端之前将需要一直等待 Beta 服务的响应，直到 Beta 服务将数据存储到数据库中。

这种做法存在两个问题。首先，Alpha 服务现在要求 Beta 服务是启动状态并且可以响应请求 - 它的正常运行依赖于 Beta 服务。其次，Alpha 服务要等到 Beta 服务的响应才能响应客户端 - 它的性能也依赖于 Beta 服务！

在这两种情况下，Alpha 服务耦合于 Beta 服务。Beta 服务失败的频率是多少？它需要停机维护的频率是多少？它的峰值性能是多少？难道真的只有在数据被安全的存储之后才可以响应，或者说提前响应就是不可行的？如果它依赖于其他服务，相应的会需要进一步延伸耦合链？像这样的系统好坏程度取决于其最薄弱（weakest）的服务。

如果我们使用 Kafaka 消息队列做为接口替换上面描述的，我们将会得到一个完全不同的结果，Kafka 有一招：一个 Kafka 消息队列是持久化存储的。当数据已安全地放到队列中时，Alpha 服务就可以响应请求了；我们可以确信数据将最终会存储到数据库中，因为 Beta 服务致力于处理队列中的消息。Alpha 服务现在仅仅依赖于 Kafka 的正常运行和性能了 - 这两个指标都可能远远好于系统中的其他微服务。它是如此松耦合于 Beta 服务，它甚至都不需要知道它的存在！

当然，消息队列从来就不是性能问题的灵丹妙药 - 它并不能神奇的改善系统的整体性能(举起手来,如果在你曾经参加的会议中有人相信的话)。然而，它确实允许你应对来自外部系统的可变负载，或者甚至是你自己系统中的微服务（例如那些必须做某种形式的批处理）。消息队列能够应对峰值负载，从而使下游服务可以平滑的速度处理。一个给定的服务的性能只需要大于系统的平均负载，而不是峰值负载。

使用 REST 接口时，你可以通过在每个服务中存储数据来打破依赖链并实现类似的效果。然而，为了达到这一点，在每一个服务中你都需要自

己实现消息代理模块。使用现有的服务你自己不必再设计、构建和维护一套（或多套）类似的系统。

服务间松耦合

在设计系统的时候，我们发现如果 Alpha 服务返回的结果是转换后的数据，一些客户可能会发现它会很方便。

如果使用 REST 接口这将变得非常容易 -Beta 服务可以返回转换后的数据，Alpha 服务直接透传给客户端。同时我们在两个服务之间增加了一个新的依赖关系 -Alpha 服务现在依赖于 Beta 服务的转换功能。这和上面小节中 Alpha 服务依赖于 Beta 服务存储数据是类似的情景。同样的问题出现了：如果 Beta 服务不能及时并且正确执行其功能，Alpha 服务同样的也不能返回结果给其客户端。

和存储数据不同的是 Kafka 针对这个问题没有提供有效的解决方案。事实上，用 Kafka 接口来达到这个目的将会更复杂（但绝不是不可能）。因为消息队列是单向通信信道，从 Beta 服务获取转换后的数据需要相反方向的第二条信道。匹配这些异步响应结果和等待的客户端也会需要一些额外的工作。

然而，用消息队列不容易做到这个事实的给了我们一个启发，那就是我们新生的系统有些地方做的并不好，我们应该重新评估。纵观我们加诸于系统的额外功能与数据流的关系，结果表明不管它是如何实现的，是该功能引入了服务之间耦合。要想让 Alpha 返回转换后的数据则他必须依赖于真正做数据转换的服务，我们架构的系统不可能绕过这个事实。

这让我们停下来检查我们是否确实需要这个功能。有没有其他的方法可以提供访问转换后的结果数据并且不会引入这个问题？客户端是否需要

所有转换后的数据？如果这个功能是必要，表明我们在切分微服务的时候并不是最优的，我们应该将数据转换这步放到 Alpha 服务中。如果不是必要的，我们只是阻止了自己添加不必要的或设计不当的功能，这些功能未来可能会影响我们系统的发展和性能。

系统的功能一旦添加上往往很难移除掉了。为了避免在系统中引入沉重的包袱，在实现之前辨别有疑问的功能（或者功能设计）是非常必要的。Kafka 消息队列的自身的局限性可以指导我们设计出更好的系统。当你试图砍掉一个紧急的功能时，他们可能会感到沮丧，但是从长远来看是值得被称赞的。

提高可用性和扩展性

随着负载的增加，我们的系统应该保持高可用性，可扩展性。作为实现这一目标的一部分，我们希望能够运行多个 Beta 实例。如果其中一个实例宕机，其他的实例会（希望）仍然能够正常运行。可以增加更多的实例来处理增加的负载。

使用 REST 接口时，我们可以在 Beta 服务实例前面加一个负载均衡器，并且把 Alpha 服务从直接指向 Beta 服务的实例替换为指向这个负载均衡器。负载均衡器需要能够自动检测宕机的实例，从而一旦有实例宕机可以将负载转移到别的实例上。

一个 Kafka 消息队列默认支持可变数量的消费者（例如 Beta 服务），不需要额外的基础架构的支持。多个消费者可以组成一个“消费组”，只需要在连接集群的时候指定一个相同的组名。Kafka 会将每个主题所有分区的数据共享传输给整个组的消费者。只要我们使用的主题有足够多的分区，我们可以持续增加 Beta 服务的实例，这么它们将会分担一部分负载。

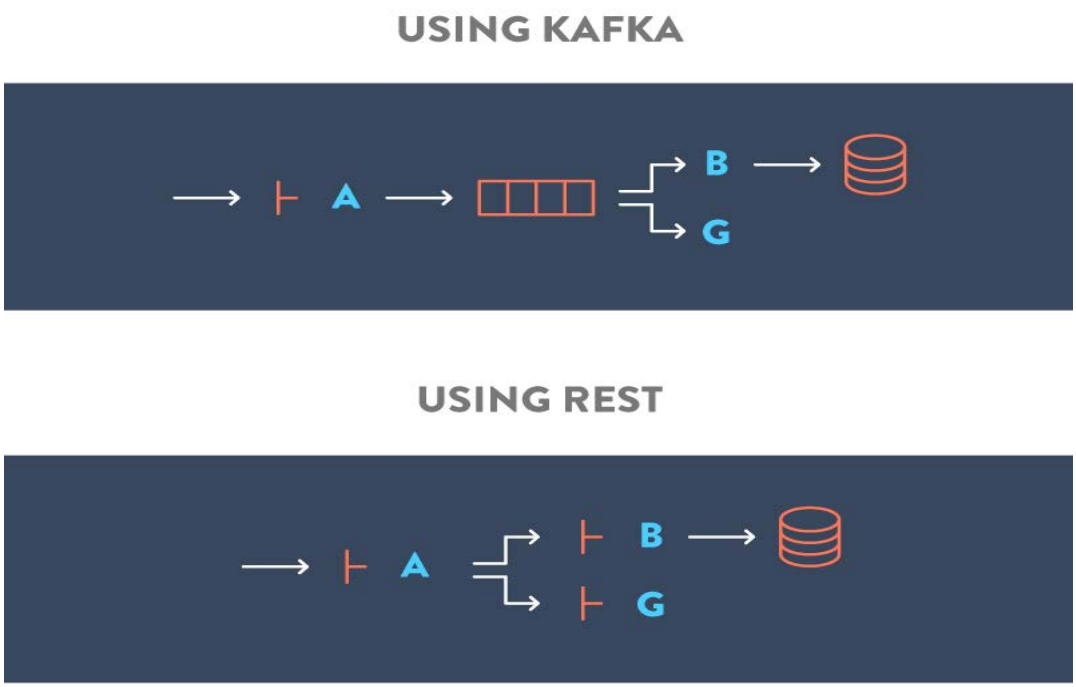


图 2

如果某些实例不可用，他们的分区将由剩余的实例处理。

增加更多服务

我们需要在我们的系统中添加一些新的功能（这并不重要），我们将把它放在一个新的 Gamma 服务中。它依赖 Alpha 服务的数据的方式和 Beta 服务依赖 Alpha 服务的数据的方式是一样的，并且数据是用同样的接口提供的。数据必须经过 Gamma 服务处理才算被整个系统完整的处理。（见图 2）

如果使用 REST 接口，则 Alpha 服务将会耦合一个额外的服务，加剧前面讨论的服务间耦合的问题。随着下游服务的增加，依赖会变得原来越多。

此外，一组数据可能成功的被一个下游的服务处理，但是在另外一个服务中却处理失败。这种情况下，可能很难通知到客户端和做到自动恢复，

特别是如果它们可以在状态不一致的情况下就离开。

使用 Kafka 接口允许新的 Gamma 服务和 Beta 服务一样简单地从同一消息队列中读取数据。只要它使用一个不同于 Beta 服务的消费组，两者之间不会互相干扰。由于 Alpha 服务不需要知道它写入数据的消息队列有什么服务在使用，我们可以持续的添加服务而不会对它造成任何影响。

数据被安全地存储在 Kafka 中，所以各个服务可以在瞬时故障的情况下重试，例如数据库死锁或者是网络问题。非瞬时错误，例如脏数据，和使用 REST 接口一样都会存在类似一些问题。检查数据合法性越早越好，例如在阿尔法服务，是减少这些问题的关键。

总结

以上的例子都是直接取自于我们在 Movio 推进微服务化过程中的真实案例。在这个过程中它已经证明了是一个非常宝贵的工具，催生了优秀的系统架构，并可以简单和快速的实现它。我们将继续探索使用它的新方法，并期望我们的使用会促进系统的进一步发展。

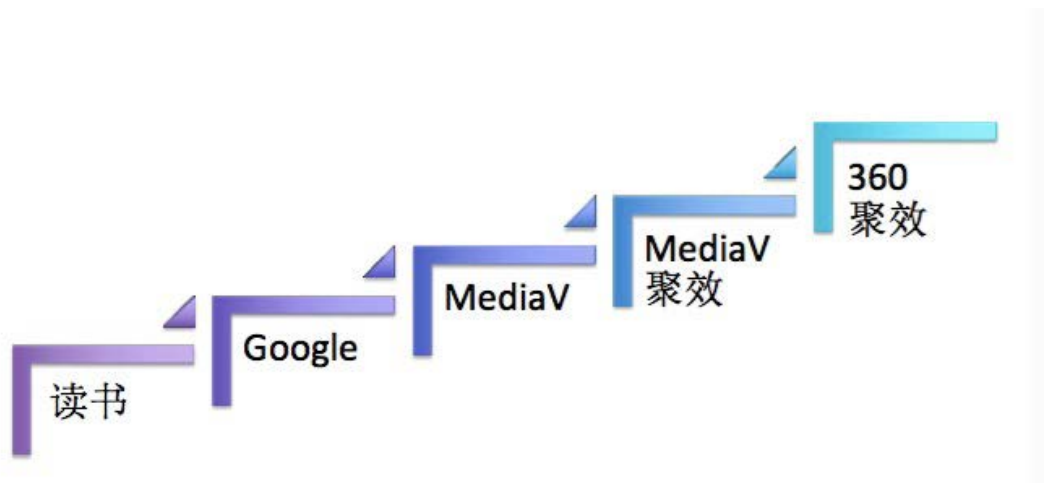
LinkedIn 和 Apache 的团队创造了一件伟大的作品。

从“人事钱心”四个方面讲述跨过技术创业的那些坎儿

作者 胡宁

我是胡宁，来自 360 聚效，今天在这个技术创业的论坛上，我演讲的主题是“跨过技术创业的那些坎儿”。我本人经历过技术创业的整个过程，从开始的选择，到中间的融资和转型，到最后的退出。现在聚效的创业阶段已经基本告一段落，回想我走过的路，有很多经验教训。今天这个论坛属于技术创业，可能与别的更技术化的论坛比起来更轻松一些，我分享的内容更多的是一些个人的经验和感悟，正好借今天这样一个机会和大家一起讨论。

那些年我走过的路



我读了很多年的书，一直读到博士，属于第三种人（语出“男人、女人、女博士”的笑话）。后来，我就觉得不要再读下去了，我需要出来工作，于是出校门直接去了谷歌。谷歌对我的影响非常大，一般公认谷歌是世界上技术水平最高的一家互联网技术公司，它的技术管理、技术标准、工作流程，乃至它的一些文化理念，对我当时乃至以后的人生都造成非常大的影响。

我在谷歌工作了五六年，然后发生了一个很大的转折。

我在谷歌做得还不错，我是少数几个升得最快的华人之一，离开谷歌之前我已经做到技术总监了。要离开谷歌的时候，有一个女同事还在问我，说你疯了，为什么要出去创业，为什么不抓紧时间结婚生子，舒舒服服地在谷歌待下去？大家知道，谷歌的福利特别好，无论是餐食，还是其他一些福利。而且作为女性，大家觉得你那么喜欢折腾，其实是蛮奇怪的。

我之所以要创业，是觉得在谷歌我学到了很多东西，我还希望发挥更大的作用。另外还有一点，虽然我在谷歌做到中层，但是始终会感觉到自己只是个螺丝钉，有什么意思呢？你对很多事情其实是没有操控力的。

讲一个简单的例子。大家知道 2010 年谷歌离开中国。宣布的那一天，我正好在回总部汇报的路上，当时谷歌北京的人都不知道这件事情，是记者打电话到我手机上采访我，说谷歌宣布退出中国，你怎么看？我才知道。

对我来说，这件事情让我非常痛苦。我和团队过去两年多的精力和时间都花在为中国打造谷歌音乐、谷歌搜索等项目，希望谷歌能够在市场上与百度有得一拼，当时业务整个数据确实还不错。但是忽然说要退出，然后我们就退出了。对中层来讲，会发现在一个大公司里面，对一个项目的成败与否，包括我是否要做这个项目，我怎么做这个项目，根本没有太多的话语权。

我那个时候就坚定了决心，我一定要自己出来做。

正好也碰到一个机会，MediaV 的负责人找到我。他原来是好耶的总裁，后来一帮人出来，成立了一个广告公司，要找一个做技术的合伙人。我原来在谷歌没有涉及广告，主要是做搜索、移动方面的事情，我觉得这个领域非常有意思，你会看到一个主要的互联网商业模式的运转逻辑是怎样的。互联网赚的钱，其实很大一部分来自于广告。我一直都说的一句话：广告是互联网行业的引擎，譬如谷歌、Facebook、阿里、百度的收入大部分都来自于广告。我非常有兴趣把这条逻辑线弄清楚。

那时，我与 MediaV 大概沟通了四个月的时间。虽然谷歌已经把我调回总部去负责安卓的云服务，包括 APP Store，我最后还是决定离开，加入 MediaV 做 CTO。

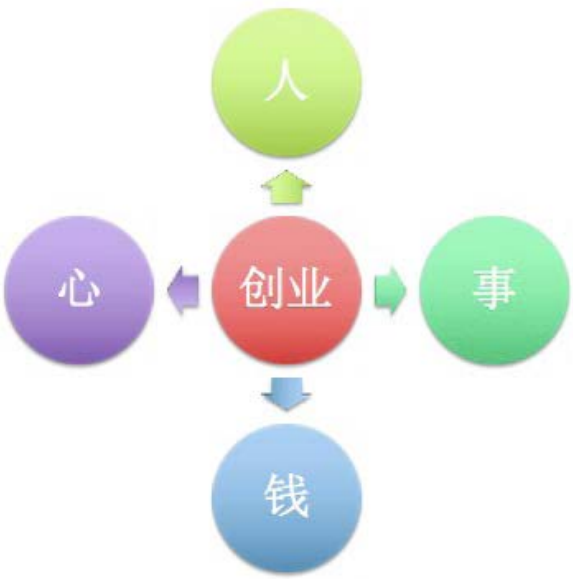
2012 年，我们推出了一个广告技术平台叫聚效。聚效发展得非常快，2013 年底我们决定分拆 MediaV 成两块。一块是聚胜万合，主要提供互联网营销整体服务，被国内一家上市公司（利欧）收购了，利欧原来是做水泵的传统公司，收购聚胜万合后开始转型，现在已经成为国内最大的互联网营销服务商之一。我们开玩笑说，水泵公司成了互联网广告公司大拿。另外一块就是聚效，主要是广告技术平台。大部分技术和产品团队都归到聚效，轻装上阵，独立发展。

2014 年，我们宣布接受了 360 的战略投资，但是我们还保持独立运营，希望上市。2015 年我们正在准备上市的时候，发生了一件事——360 宣布要私有化，要回国来上市。我们当时有点尴尬，因为 360 跟我们是一个母子公司的关系，可能会造成业务的竞争，导致我们没法上市，或者 360 没法上市。最后我们合计了一下，决定整体并入 360，就是 360 整体收购我们，我们进去以后负责 360 商业变现的产品和技术。

这一路下来，我这些年的创业从刚开始的加入，到融资（MediaV 融了很多轮，从 A 轮到 C 轮一共融了差不多七千多万美金，是国内广告技术行业里面融资量非常大的一个公司），到分拆，然后到退出被 360 收购，这整个流程我都经历过，里面有很多的经验教训。

创业的四个方面

创业可以从四个方面来看，第一是人，第二是做的事情，第三是钱，第四就是心。这四个方面我都会详细展开来讲。



人的因素

首先，团队肯定是创业的根本。我们现在回过头去看，为什么说投资人看很多的早期项目，他看的反而不是项目，而是人。因为我们会发现一个公司做了一段时间以后，往往 90% 以上的可能性，它做的东西跟它最初想要做的项目完全不一样，可能连方向都不一样。在这种情况下，如何保证成功？关键就在于人。

1、找合伙人

大家可能听说过一个普遍的说法：我有一个很好的 idea，我就等一

个靠谱的技术合伙人。这些年里，有无数的企业、无数的人托我介绍一个靠谱的 CTO，这个 CTO 好像是大家普遍紧缺的一个岗位。但大家可能并没有真正了解到，这个 CTO 到底能做什么？而这个 CTO 对合伙人团队的要求又是什么？对于一个技术人员来说，如果你要技术创业，很可能你的岗位就是 CTO，你对合伙人的挑选我觉得要考虑几个方面。

第一，你怎么认识这个合伙人？有若干途径，可能是你同事，或者之前的员工，或者投资人介绍，或者是现在有种服务，是介绍搭配合伙人的。从渠道来讲，你越熟识越好，因为知根知底。

另外，人品非常非常重要，尤其是在创业的时候。为什么重要呢？是因为你在创业的过程中会碰到无数利益相关的事情。就人性来讲，不管他开始有多好，当赤裸裸讲到钱，特别是利益相关的时候，很多人会露出他的真面目。人品里面我觉得分两部分，一部分是他觉得什么事是 OK 的？什么事是不 OK 的？在这儿我们不做道德判断，每个人的道德底线是不一样的，但最好跟你是一致的。

我举一个例子。比如说你的竞争对手出了一个新产品功能，还不错，如果你的合伙人认为我们要抄袭，而你认为，我们不应该抄袭人家。如果你们两个在这件事上有巨大的分歧的话，其实最后都是不能长久的。这就回到什么事能做，什么事不能做？这个是探测大家的底线是否是一致的。

第二，合伙人是否重承诺。承诺同样也是非常非常重要，尤其是对于技术人员来说。技术人员相对来说比较单纯，开始的时候人家说什么就容易信什么，真正创业的时候，你会发现你所处的环境比你在大公司里面相对来说比较单纯的环境会复杂很多。比如，一些朋友去一家创业公司，本来承诺有股权，进去以后可能长达半年都不签股权协议，一问就推托，一问就推托。这就说明这家创业公司没有办法兑现承诺。这种情况我一般的

建议是，尽早离开，因为待的时间越久，沉没成本越高。

第三，合伙人最好整个资源和背景跟你是互补的，这个也比较好理解了，我当时决定加入 MediaV 实际上也是因为其他几个合伙人都是销售出身，我对销售工作不熟悉，我对国内的广告市场也不大了解，有这样的合伙人跟我互补的话，团队实力就会比较强。

第四，你和合伙人的个性是否比较兼容。

列完了这几点要求，我们会发现，其实就是像寻找另外一半一样去寻找你的合伙人。想一想未来这几年，你跟他待的时间、跟他说的话、跟他去做的事情、要面对的困难，可能比你的伴侣跟你在一起待的时间、说的话、面对的困难还要多，所以一定要谨慎。

2、招聘

招人对于所有创业团队来说都是很头疼的事情。最简单第一点是，我怎么设定我的标准。现在有两种标准，一种标准是其实只要招差不多的人就可以了，因为要做的事情这么多，大概就招一个可能比如说五千块钱、八千块钱的人就行了。还有一种是要找最好的人。

这两种方式都各有优点和缺点。比如说招很多水平一般的人，好处是我能解决目前的一些困难，但是问题是他的管理成本会直线上升，而且人，特别是优秀的人，有一个倾向性，优秀的人愿意跟优秀的人一起工作，你招的一大帮一般的人，优秀的人就进不来了。

我自己学到的是，不要降低标准，你一旦设定了这个标准，就严格按照这个标准执行。曾经有一位博士面试后不符合我们的要求，但他一直在求我们，我就心软了。第一我觉得他态度确实还挺不错，表示愿意学，第二我觉得读了很多年博士也不容易，就让他进来了。结果，进来后半年很痛苦，他学习的态度很好，学习的速度却很慢，达到的结果也很差，最后

我们不得不让他走。这件事对我们、对他来说，都是一个惨痛的教训。

作为一家创业公司，可能并不是特别有名，为什么人家不选择百度、腾讯、阿里，而是来加入我们呢？我们招到的很多人才，在招聘面试的时候跟我们的工程师或产品经理聊，发现原来我们专业性非常强，而且公司里的人都很厉害，最后心甘情愿来到我们公司。招聘实际上是一个很好的展示平台，你们要善加利用。

最后讲一句大实话：招你能找到的最好的人。因为我之前也试过招一般的人，我会发现其实性价比最高的，或者最省事的就是招你能找到最好的人。

3、管理

技术管理、公司管理涵盖的内容很多。例如管理层级，你到底需要一个相对比较平的层级，像谷歌那样的，还是一个典型的中国的大公司，中国的大公司一般都是层级森严。不同的管理层级有不同的优点：当层级很平的时候，容易激发人的创新性，谷歌的创新很多是来自于底层。而层级有效建立起来的话，令行禁止，执行力会很强。在创业阶段，我的建议是，在早期要比较平的层级，在后期你可以慢慢建立起层级。

一些创始人是事无巨细都要一把抓，在早期是可以的，但团队发展到几十人的时候就不行了，你需要能够去放权，你需要能够培养骨干。我个人的习惯是，搭建团队的时候首先去找那些骨干、有可能成为骨干的人、有潜力的人，把他放到这个位置上，花很多精力去教他们培养他们。这些精力最初看起来是浪费，实际上不是，只要把他教出来了以后，他能带领他下面一整个团队。最终你这个架构就很健康。

还有目标考核。最近有一个争议，就是史玉柱提出的兔和狼理论，他抱怨下面好多人都是兔子，他要淘汰，要培养狼性。我觉得这个本质问题

应该是老板的问题。为什么是老板的问题？很简单，如果你没有把目标设定清楚，没有把相应的考核和奖惩设定清楚，下面的人就像小兔子一样茫然失措，搞不清方向。所以目标考核非常重要，只有大家都明白是往哪个方向前进，才能有巨大的驱动力。

文化理念说起来很虚，其实都体现在细节里面。我们讲到工程师文化、产品文化，体现在公司对待工程师、产品经理是否细节上做到尊重。我刚进 MediaV 的时候发现工程师用的电脑显示屏是 15 寸的，就马上要求所有工程师的显示屏都换成 22 寸以上的显示屏。其实没多花多少钱，但是对于提高工程师的效率，给予他们的尊重会感觉好很多。

文化理念也都渗透在一点一滴的细节里面。我不认为你应该喊口号，喊口号没有什么太大的用处，最关键的是你在创业的时候，是否你自己相信这一点，比如说工程师驱动，或者是产品驱动，体现在决定一件事情的时候，谁有最后的话语权。公司的文化就是按照这样的理念建立起来的。

事情的因素

简单来说，创业就是做好一件事。

1、方向和模式

首先是方向和模式，俗话说的是风口。大家可能都知道雷军说的一句话，在风口上猪都能飞起来。但你能看到一个风口，往往别人也能看到。我们往往看到，很多方向一旦被看好，马上有无数的人挤进去，迅速变成红海。最后能够拼杀出来的人都是踩着累累尸骨站起来的。所以，找风口未必是件好事。

其次，你如果寻求一个方向，是做 2C 的产品，还是 2B 的产品？这两种方向我都做过，都有比较深刻的感受。2C 产品面向最终消费者，比较

好理解，但自然也竞争非常激烈，除非你能快速达到自然增长点，否则的话就会很快死去，所谓“早死早超生”。2B 是面向企业的，比如做广告营销就是典型的 2B 的企业，相对来说 2B 基本没有盈利模式的问题，因为它可以向用户，即企业收钱，所以一般 2B 企业只要抓住了几个企业客户，就能活下去，所以 2B 企业的运营时间一般要比 2C 公司长很多。但 2B 公司做久了以后，成长性会出现问题。它成长没有那么快，因为在一个企业市场，或者行业市场里面，成长空间只有那么大，不会像微信那样恐怖地增长，往往到了一定程度就像鸡肋一样，食之无味弃之可惜。

最后，是小而美和大而强。小而美的项目，其实也是非常好的。可能大家都希望把事情做得非常大、非常强，这当然跟融资也相关，投资人追求成长性，追求规模化，推动大家都往大而强走。但如果能够把一个小而美的项目做成，其实是一件非常好的事情，也能为社会造福。这也是一个选择。

后面两点，盈利模式与 2C 相关，行业经验与 2B 相关。如果面向企业的话，你一定要有非常深厚的行业背景，否则你不知道痛点在哪里。我认为很多时候找方向、找模式，与其找风口不如找痛点。如果你在这个行业有很深厚的经验，也做了很多客户的调研，你发现这是个痛点，大家其实都有这个需求，但是都没有现成的产品去满足这样一个需求的时候，我认为找到这个痛点比找风口要靠谱得多。

2、从想法到产品，即执行

我的 PPT 上面列出了原型验证、聆听反馈、快速迭代的条目，其实就是“精益创业”的原则，推荐大家可以去读这本书（《精益创业》），书里讲的很多原则都非常适用于创业公司。大家如果在大公司里面工作过，可能有一个倾向是，我要拼命要资源，因为会哭的孩子有奶吃，而资源要

的多能干的事情也多。但是对于创业公司来说，人手是非常紧缺的。创业公司里面人很少，外面的竞争也很激烈，需要的策略就是单点突破。你只要做一到两个对用户来说是最急需的、最好的功能就成了。你不要想大而全，这对创业公司来说是非常忌讳的一件事情。天下武功，唯快不破，创业公司讲白了就是快。大公司从资源上来讲是比创业公司要多很多的，而且执行力比创业公司也更强，但大公司里面的项目决策流程相对来说是比较慢的，从立项到调配人手、到执行，整个流程都非常慢。

3、研发

对于研发，我的一个感受是，研发流程你要自己做过一遍，你才知道为什么是这个道理。比如，瀑布流模型和敏捷开发，为什么敏捷开发对于互联网产品研发会更有效？用不同模式做过一遍就知道是什么缘故。而且敏捷开发里面，哪些是比较好用的，哪些是不好用的，我觉得强实践才会出真知。我认为管理理论就是经验科学，你只有自己做过了以后，才能找到你自己认为最有效的方法。所以不能只听别人总结的很多各种各样的理论、各种各样的原则，最终还是要你自己去做大量的实践。

钱的因素

资金是创业成功的关键。

1、股权分配

首先，就是股权分配。大家希望自己创业成功以后能够财务自由，股权分多少就决定了最后是否能财务自由。但是股权怎么分？一般来说，如果你是与一个人或几个人一起去创业，最关键、最主要的人，比如 CEO，最好是拿大股。原因很简单，因为在创业的过程中肯定会产生分歧，产生分歧的时候一定要有一个人出来拍板，这个时候拍板往往是靠分配比例来

说了算。

其次，要注意看股权操作细节。你如果加入一个公司，很多时候股权协议的细节隐藏了问题。我有一个朋友跟他老板去开一个公司，他老板承诺给了很多股份。去了以后发现他老板给他签的股权协议里的那个公司并不是他们的母公司，而是底下的一个子公司。这实际上就会造成什么问题呢？如果未来谈到退出、利益分配的时候，很可能会把这个子公司本来的权益挪到其他的子公司去，这个协议就是废纸一张。

股权内容对于很多技术创业者来讲有很多陷阱。我刚才讲到技术人员还蛮单纯的，很多操作手法他不知道。有可能你辛辛苦苦工作了好几年，最后发现股权协议就是废纸一张，这是出现过的。

股份如果没分好，一定会打架。所以这件事情你一定得谈，而且早谈比晚谈好，不要扭扭捏捏。我觉得很多技术人员会有一种士大夫脾气，不好意思跟人讲讨价还价，不好意思跟人讲钱，不好意思讲你应该分给我多少股份。其实你迟早要讲，早讲比晚讲好，要理直气壮地讲。

2、资金管理

创业的时候，你会特别敏感钱都是怎么花出去的。第一个方面就是人力成本，尤其是研发成本会非常高。因为工程师、产品经理都非常非常贵，属于市面上最贵的一类人才，研发成本如果你不控制住，这个花出去如流水一般。

还有一个更重要的方面就是市场营销。很多公司融了很多钱，最后为什么失败了？大部分的钱都被市场营销花走了，而且花的不是地方。

我现在做广告，发现很多公司蛮蠢的，他花了这么多钱能带来多少新客，这些新客在其生命周期里面会带来多少的收入，他没算过这个账，人家一忽悠说这个不错，他只觉得看上去显得很高大上，就把这个钱花出去

了。前一段时间，O2O 很多钱就花得莫名其妙，砸大量的钱去线下送礼物拉注册，而这种注册的质量都很差。关于市场营销，你即使是属于 CTO 这样的角色也要注意，因为这是一个公司的命脉，大量的资金是从这里流出去的。要关注相关的数据，要关注它的效果。

用一句话总结，就是现金为王。

3、融资

融资我们也做过好几次，还是有挺多的经验或者教训的。首先是融资时间点。正常来讲，你至少要提前半年去融资，因为融资不可能在短时间内做完，来来回回会耗费很多的时间。从拿 term sheet，到做 DD，到最后真正打钱入账，需要很长的时间，至少几个月。其次是控制权问题。去年不是有合并潮吗？无论是去哪儿、携程，还是之前更早的土豆、优酷，为什么合并能成功？其实不是创始人团队去推动的，它是投资人去推动的。原因很简单，当融了太多资以后，创始人团队失去了对公司的控制。很多投资基金一轮是 5~6 年，差不多年限后基金就要着手退出投资，要交付他们的 LP 多少回报，所以投资人不可能长时间持有一个公司的股份，而永远不退出这个公司。所以到一定年限了以后，投资人会迫使公司退出。如果这个时候所持股份比例太低，创始人就会失去对自己公司的控制。

所以，你融完资以后一定会碰到这个问题，我怎么退出？刚才说了 VC 一般五六年后就要退出。另外，对于互联网创业公司来说，一般工作了大概三四年、五六年以后，如果还不退出，整个团队很多人心里会犯嘀咕。因为所有互联网公司时间都不长，谷歌、Facebook 最长的也就十几年，三四年、四五年对一个互联网公司来说已经是不短的时间了，如果还不上市，或者还不退出，你的团队可能会出现一些不稳定的因素。

退出的选择有两种，一种是上市，一种是卖掉。

其实上市对于创始人团队来说，并不是一个退出。上市之前需要做很多的努力，达到上市所需的各种指标，还需要按要求维持一段时间。上市后，还要为每个季度的季报做努力，永不停息。

卖掉稍微简单一点，但如果是要卖掉的话，你也要考虑，怎么样卖掉能够使得整个团队得到一个妥善的安排。

融资有一点要注意的，叫优先清偿权。大家如果做过融资的话，可能会知道这个概念。这个概念很重要，因为退出的时候，你会发现因为投资人有优先清偿权。例如投资人投了两千万，他会把两千万乘以一个 1 以上的系数，先把这些钱拿走，剩下的钱才会由创始人团队和投资人按比例分。我一直认为这个优先清偿权并不合理，但这是现在行业的标准。这个意味着什么？如果你融了很多钱，最后你会发现即使能退出，很多创始人团队拿不到多少钱，正因为钱都给投资人拿走了。

所以我的一个结论是，不要融超出你所需的钱。我们经常看到新闻，说什么什么公司估值多高、融了多少钱，大家都很羡慕。但从长期来讲，这些团队我认为有时候是有苦说不出。估值这个东西是一个双刃剑，估值高有两个风险。

- 第一，优先清偿权，你会卖掉或者上市，投资人会拿到一大笔钱，最后创始人不见得分多少。
- 第二，要维持这么高的估值，你需要做很多的甚至超出原来这个项目所能承载的努力。一个公司原本是典型的小而美，但因为估值太高，创始人不得不去延伸方向，做他们不一定擅长的事情，努力使营收能够符合高估值对应的成长曲线。这些公司最后都做得非常非常辛苦，但又没法退出，因为价格太高了，最后有点进退维谷。

心的因素

创业会遇到的最大问题是，你会面对很多的不确定性，因为在任何一个时间点上，你可能都要做出很多的选择，而你并不确定你这个选择对不对。但是你身上有很多压力，尤其是创业到后来，公司有几百号人，甚至上千人，你的压力会非常大。我跟一些创始人朋友一起聊，大家有一个共同的问题，就是创业到了一定时间以后睡不着觉，失眠，焦虑。你还没法跟你员工说，你要对你员工打鸡血，你要告诉他公司发展得非常好。但是里面很多的不确定性，很多的后悔，你都只能自己默默地去面对。

然后就是学习。需要能够从失败里去学习。很多成功的案例更广为人知，但我认为成功有很多偶然的因素，而失败其实可以学到更多。创业有所谓的“18 个月门槛”，就是一个企业发展到了大概 18 个月会碰到一个坎儿，如果跨过去了，这个企业就活下来了，如果没跨过去，这个企业就死了。我创业的时候，真的发现有这个 18 个月门槛，真的还挺准的。原因我觉得是，企业发展到这个时候，已经慢慢成型了，团队也成型了，原来那种新兴企业蓬勃向上的势能也有所磨损，一些非常难的问题就开始浮现出来，比如说我到底怎么盈利？我的未来在哪里？

做企业做到后来会相信“谋事在人，成事在天”，这个创业最后成败与否，你只要尽力了，这个就交给上天了。

Q&A

提问：我是国内某上市公司某部门的研发负责人，听您介绍，你是 CTO，正好有个问题想问一下您。作为一个 CTO 来说，到底是技术更重要，还是管理更重要？作为一个部门的研发负责人来说，如何控制成本？您也提到了研发成本的控制，最终一个项目工作量如何衡量，是偏多还是偏少？

胡宁：第一个问题讲的是 CTO 的定位问题。有两个角色，我这里澄清一下。一种角色叫 CTO，另外一种角色叫技术副总裁，或者是工程副总裁，两个要求是不一样的。我觉得对于 CTO 来说更重要的是，你要决定这个公司的技术方向、产品方向。如果你是技术副总裁、产品副总裁或工作副总裁，你主要的职责是执行，就是管理，这两个角色是不一样的。

第二点，你讲到成本控制问题。我记得冯大辉曾经说过一句话，技术的作用往往短期看被高估，长期看被低估。其实也很简单，研发部门对一个公司来讲，它就是一个纯投入、纯消耗部门，虽然是公司的基础，但不一定能跟收入直接挂钩。成本意识对于 CTO 或者是技术副总裁来讲，需要非常非常执著。你不能是说，从大公司出来有奢侈的习惯，或者刚融了资，就要拼命招人。还是要单点突破。

首先你要跟你们公司的其他合伙人一起去算，我未来这一段时间里面，我能花多少钱。以此推算我能招多少人，在这么多人的基础上我能做多少事情。如果我做不了我想做那么多事情，怎么办？按优先级排，把一些不必要的砍掉，只做最重要的点，所以研发成本控制就是有多大锅下多少米。研发成本花起来是没有上限的。我们可以看到一个大公司里面，很多部门经理、部门总监，你会发现他不断地要人，说缺人，当然他缺人，因为预算和成本不在他的考量范围之内。但对于一个创业公司 CTO 来说，就一定要考虑成本，就一定要做平衡。这其实就是现金流的管理，最终是要创始人团队一起做决策的。创业公司现金流为王，你不可能招一个特别庞大的队伍，什么都做，否则会死得非常快。

AWSome Bar

7月相聚在深圳

ArchSummit邀你来开技术趴

2016年7月15日（周五）19:00–21:00
深圳·华侨城洲际酒店

Lean Coffee（规模 100人左右 / 免费）

亚马逊 aws

Geekbang. 极客邦科技 | InfoQ

北京 Beijing | 伦敦 London | 纽约 New York | 里约热内卢 Rio De Janeiro | 旧金山 San Francisco | 圣保罗 Sao Paulo | 上海 Shanghai | 东京 Tokyo

QCon [上海站]

全球软件开发大会2016 | 宝华万豪酒店 10.20~22

主办方 Geekbang 极客邦科技 InfoQ

Twitter监控系统如何处理十亿量级metrics

黄 浩

Twitter高级工程师

Spotify高可靠事件交付系统的设计与运维

Igor Maravić

Spotify架构师

美团大众点评微服务实践-服务框架Pigeon的设计与实现

吴 湘

美团大众点评
基础架构中心上海侧负责人

构建可伸缩的软件开发团队

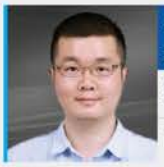
李智慧

宅米CTO

Vue 2.0: 渐进式前端解决方案

尤雨溪

Vue Technology LLC 创始人
前端框架 Vue.js 作者



外卖物流配送的大数据创新实践

蒋 凡

百度外卖高级研究员

携程的推荐及智能化算法及架构体系实践

于 磊

携程基础大数据产品团队总监

资深工程师如何胜任管理岗位——Spotify工程管理经验谈

Sebastiano Armeli

Spotify工程经理

如何选好技术初创风口：从0到1，1到100

孔令欣

点融网CTO

代码安全新突破，应用安全新战略

刘再耀

OneAPM 安全技术总监

面向IoT的云端架构设计实践

周爱民

南潮（ruff.io）架构师

以创业的思维经营开源项目

韩 卿

Kyligence Inc 联合创始人 & CEO

京东大促之大规模分布式监控系统实践

鲍永成

京东云平台资深架构师

如何打造一个百万亿级的日志搜索引擎：Poseidon

魏自立

360 高级工程师 & 资深顾问

ServiceWorkers——未来Web应用的基础API

陈 涌（题叶）

饿了么前端工程师

8折优惠 8月21日前报名 **立减1360元**

扫码关注QCon官微，了解大会最新动态

ArchSummit
全球架构师峰会2016

2016

2016年7月15日-16日

深圳站

扫描二维码了解大会

架构师 月刊 2016年6月

本期主要内容：谈谈后端业务系统的微服务化改造，新浪微博混合云架构实践挑战之不可变基础设施，分布式MySQL集群方案的探索与思考，为什么我不再使用MVC框架，AWS S3：不仅仅是存储

云生态专刊 06

《云生态专刊》是InfoQ为大家推出的一个新产品，目标是“打造中国最优质的云生态媒体”。

顶尖技术团队访谈录 第六季

本次的《中国顶尖技术团队访谈录》·第六季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。

架构师特刊 **揭秘京东618背后的技术力量**

每年6月18日是京东店庆日。在店庆月京东都会推出一系列的大型促销活动，以“火红六月”为宣传点，其中6月18日是京东促销力度最大的一天。一度将京东618促成与双11遥相呼应的又一大全民网购狂欢节。